HOW TO PROGRAM YOUR MSX COMPUTER LIKE A PROFESSIONAL

Tim Hartnell

Interface Publications
LONDON · MELBOURNE

How to Program your MSX Computer like a Professional

Tim Hartnell

First published in the UK by:
Interface Publications,
9–11 Kensington High Street,
London W8 5NP

First printing October 1985

ISBN 0 947695 28 1

Books published by Interface Publications are distributed in the UK by WHS Distributors, St. John's House East Street, Leicester LE1 6NE (0533 551196) and in Australia and New Zealand by PITMAN PUBLISHING. Any queries regarding the contents of this volume should be directed by mail to Interface Publications, 9–11 Kensington High Street, London W8 5NP.

Cover art – David John Rowe
Cover design – Richard Kelly
Printed and Bound by Short Run Press Ltd, Exeter

Contents

Foreword

You've bought a great computer, and this book will show you how to get the most out of your machine, quickly and easily. In the book, I've assumed that this is the first time you've learned to program, and that your MSX computer is the first one you've handled. Therefore, if I start talking about material you already know, just skip over it.

If you've had previous experience with programming in BASIC, you may want to turn straight to those sections which show you how to make the most of your MSX machine using its unique sound and graphic capabilities. We'll be looking at such things as creating weird sound effects and music, animated 'arcade' displays with user-defined sprites, and much more. You'll see, if you glance through the book, that there are a number of major programs ready to be typed in and run. These include the board games **Othello** and **Checkers**, plus animated graphic games like **Monster Chase** and **Multi-Sprite Zap Out.**

This book is intended to be used as a work book, to be read through with your computer turned on. To get the most value out of the book, you should enter each example program as you come to it.

I've assumed that you've followed the instructions which came with your machine which show you how to connect your computer to the television and to a cassette recorder. Although there is little point in saving the short demonstration programs, there are many of the longer programs you'll want to keep on tape. If you do this, you'll find you have quite a good library of programs by the time you finish this book.

Once you've followed the instructions which came with the machine which tell you how to hook it up to the TV and cassette recorder, turn the page and let's get on with the fun and action.

Tim Hartnell,
London, 1984.

Tim Hartnell is one of the most widely-published computer authors in the world. Recent books include **Tim Hartnell's QL Handbook** and **The Big Fat Book of Computer Games.** He is also the author of the acclaimed work **Exploring Artificial Intelligence on your Microcomputer.** The book includes 13 major 'intelligent' programs which will run on your MSX machine.

CHAPTER ONE

Putting Things on the Screen

We start learning to program using the most commonly-used command in BASIC, the word PRINT.

Type the following on your computer:

```
PRINT 2
```

Until you press the RETURN key, the computer will do nothing. Specifically, at this point, it will ignore the command PRINT 2. Press RETURN now, and you should see the number 2 appear underneath the words PRINT 2. This is the way PRINT works. It takes the information which follows the command PRINT, with a few exceptions which we'll learn about shortly, and PRINTS this on the screen which is, after all, exactly what you'd expect it to do.

But your computer is not completely stupid. That is, it can do more than just blindly print what you tell it to. If the word PRINT is followed by a sum, it will work it out before printing, and give you the result. Try it now. Enter the following line, then press RETURN:

```
PRINT 5 + 3
```

You should see the figure 8 appear. The computer added 5 and 3 together, as instructed by the plus (+) sign, then printed the result on the screen. It can do subtraction as well (clever inventions, these computers). Type in this, and press RETURN to see subtraction (and PRINT) at work:

```
PRINT 7 - 2
```

Now the computer can – of course – carry out a wide range of mathematical tasks, many of them far more sophisticated than

simple addition and subtraction. But there is a slight hitch. When it comes to multiplication, the computer does not use the × symbol you probably used at school. Instead, it uses an asterisk (*) and for division the computer uses a slash (/).

Doing more than one thing at once

The computer is not limited to a single operation in a PRINT statement. You can combine as many as you like. Try the next one, which combines a multiplication and a division. Type it in, then press RETURN to see the computer evaluate it:

```
PRINT 5*3/2
```

This seems pretty simple. Just type in the word PRINT, follow it with the information you want the computer to print, and that's all there is to it. But, it is not quite as simple as that! Try the next one and see what happens:

```
PRINT testing
```

That doesn't look too good. Instead of the word testing we've got a zero. The computer thought we wanted a 'variable', rather than the word testing. We won't try to explain the meaning of the word variable at this point (it's not on the curriculum for this chapter), but it means simply that the computer thought you wanted to print a number which had the name of testing. Foolish machine. Computers may be very, very clever machines, but they need to be led by the hand, like a very stupid child, and told exactly what to do. Give them the right instructions and they will carry them out tirelessly and perfectly, without an error. But give them incorrect instructions, or – even worse – confuse them, and they give up in despair, or do something quite alien to your intentions.

Strings

If you want the computer to print the word testing, you must put quote, or speech, marks around the words, like this:

```
PRINT "testing"
```

This time when you press the RETURN key, the word testing will appear at the top of the screen. This is worth remembering. When you want the computer to print out some words, or a combination of words, symbols, spaces and numbers, you need to put quote marks around the material you want to print. Information held in this way between quote marks is called a most peculiar name in computer circles. The jargon for the information enclosed in quote marks is 'string'. So, in our example above, the word testing, when enclosed in quote marks, is a string. You can, in fact, get away with just the first quote mark so the line reads PRINT "testing but this is not good practice.

Our first program

Type the following into your computer. Notice that each line starts with a number. Type this into the computer, and follow this with the other material.

```
10 PRINT "Jack and Jill"
```

Type in the next line, the one starting with 20, and press RETURN once you have it in place. You'll see, by the way, that your MSX computer automatically turns words like PRINT into capital letters when the program is listed out, even if you entered them in small letters. Text within quote marks is not changed from small to capital letters by the computer. Now enter the rest of the lines in this program:

```
20 PRINT "went up the hill"
30 PRINT "to fetch a pail"
40 PRINT "of water"
```

When you run this you'll see the following (if all is well):

```
Jack and Jill
went up the hill
to fetch a pail
of water
```

Adding new lines

The computer, clever beast that it is, allows you to enter your lines in any order you choose. It will then sort them into order for you. Although our first program, and most of the other ones in this book, are numbered in 10's, starting at 10, there is no particular reason why you should follow this convention if you do not want to. However, there is a reason for leaving 'gaps' in the counting. Although our first program could easily be numbered in 1's, it would leave no room to add later lines, if we decided there was a need to do so.

To see the computer sorting lines into order, adding the following:

```
25 REM a line in the middle
```

Now type in LIST, to get the computer to list out the current program it is holding, and you'll see line 25 neatly in its proper numerical place. Now, run the program again. You should find that line 25 made no difference at all to it.

Making remarks

Why did the computer decide to ignore line 25? The word REM stands for remark, and is used within programs when we want to include information for a human being reading the program listing. You'll find REM statements scattered throughout the programs in this book. In each and every case, the computer ignores the REM statements. They are only there for your convenience, for the convenience of the programmer, or of someone else reading the program.

Often you'll use REM statements at the beginning of the program, like this one:

```
5 REM Jack and Jill poem
```

You may wonder why this would be necessary. After all, it is pretty obvious that the computer is holding a 'Jack and Jill poem', even without the line 5 REM statement. You are right. In this case, there is

little point in adding a title REM statement to this program. But have a look at some of the more complicated programs a little further on in this book. Without REM statements you'd have a pretty difficult time trying to work out what the program was supposed to do.

REM statements are often scattered throughout programs. They serve to remind programmers what each section of the program is supposed to do. Once you've been programming a while, you'll be amazed at how many programs you'll collect in listing form which – when you go back to them in a month or two – will seem totally obscure. You won't have a clue how the program works, or even more important, what on earth it is, or what it is supposed to do. This is where you'll find REM statements invaluable.

It is worth getting into good habits early as a programmer. So, I suggest you start right now adding REM statements to programs. If you come across programs, or program fragments, in this book which you want to keep, and which do not have REM statements, get into the habit of using them by adding REM statements to these programs. And make sure you use them in your original programs.

Back to PRINT

Let's return to the subject of the PRINT command. Empty your computer's memory by entering NEW and then pressing RETURN. Type the following program into your computer and run it:

```
5 CLS
10 PRINT 1,2
15 PRINT
20 PRINT 1;2;3
25 PRINT
30 PRINT "MSX computer"
35 PRINT
40 PRINT "23 + 34 = ";23 + 34
45 PRINT
50 PRINT 2*3
55 PRINT
60 PRINT 3^5
65 PRINT
70 PRINT "The answer is ";23+5-7/6
```

There is a lot we can learn from this program.

Firstly, as in the Jack and Jill program, the computer executes a program line by line, starting at the lowest numbered one and proceeding through the line numbers in order until it runs out of numbers, when it stops. (You'll discover that this orderly progression of line numbers does not always apply, as there are ways of making the computer execute parts of a program out of strict numerical order, but for the time being it is best to assume that the program will be executed in order.)

Look first to line 10 of your program. You can see that there is a comma between the 1 and the 2. This has the effect of making the computer print the numbers with a wide space between them. The comma can be used in this way to space out numbers neatly for a table of results or a similar purpose. (Try PRINT 1,,2 and see what effect this has.) When you use a comma in this way, to divide the things which follow a PRINT statement (but not when the comma is inside a string, that is, between quote marks) you'll find it divides the screen up into neat little rows. Try PRINT 1,2,3 and see the result of the commas. Then you can try the effect of PRINT 1,,,2,,,3,,,4,,, 5,,,6,,,7,,,8,,,9,,,0 to make it perfectly clear what is going on.

The third line of the program, 15, is just the word PRINT with nothing following it. This has the effect, as you can see in the display on your screen, of putting a blank line between those lines which include material after the word PRINT. The same comment, of course, applies to lines 25, 35, 45, 55 and 65.

Line 20 has three numbers (1, 2 and 3) separated not by commas (as in line 10) but by semicolons (;). Instead of separating the output of the numbers as the comma did, you'll see that it causes them to be printed with a single space on either side of them. When printed, numbers are always followed by a space. Positive numbers are also preceded by a space. You use the semicolon when you want printed material to follow other printed material without a break.

Line 30 is a word, and this is a ... If you mentally said 'string' when you came to those dots, then you're learning well. This word is a string, in computer terms, because it is enclosed within quote marks.

Line 4Ø is rather interesting. For the first time we have included numbers and a symbol (=) within a string. As you can see the computer prints exactly what is within the quote marks, but works out the result of the calculation for the material outside the quote marks, giving -- in this case -- the result of adding 23 to 24. Try to remember that the computer considers everything within quote marks as words, even if it is made up from numbers, symbols, or even just spaces, or any combination of them, while it counts everything that is not within quotes in a PRINT statement as a number. This is why it got so upset earlier when we told it to print the word testing without putting the word in quote marks. It looked for a number which was called testing and because it could not find one (as we had not told the computer to let testing equal some numerical value), it refused to co-operate.

So line 4Ø treats the first part, within quote marks, as a string, and the second part, outside quote marks, as numerical information which it processed.

In line 5Ø we see the asterisk (*) used to represent multiplication and the computer quite reasonably works out what 2 times 3 is and prints the answer 6. In line 6Ø we come across a new, and strange sign, ^. This means 'raise to the power' so line 6Ø means print the result of 3 raised to the fifth power. In ordinary arithmetic, we indicate this by putting the 5 up in the air beside the three. However, it is pretty difficult for a computer to print a number halfway up the mast of another number, so we use the ^ symbol to remind us (by pointing upward) that it really means 'print the second number up in the air'.

The final line of this program combines a string ('the answer is') with numerical information (23 + 5 – 7/6). You can see that, as expected, the computer works out the sum before printing the answer, and prints the string exactly as it is. Look closely at the end of the string. You'll see there is a space there. After the closing quote there is a semicolon which, as we learned in line 20, joins various elements of a PRINT statement together. This semicolon means that the result of the calculation is printed up next to the end of the string.

This brings us to the end of the first chapter of the book. I'm sure you'll be pleased at how much you've learned so far and are looking forward to continuing your learning. But now you've earned a break. So take that break and then come back to the book to tackle the second chapter.

CHAPTER TWO

Ringing the Changes

It's all very well getting things onto the computer's screen as we learnt to do in the last chapter, but from time to time you'll discover we need to be able to get printed material off the screen during a program, to make way for more PRINT statements. We do this with a command called CLS, for CLear the Screen.

Clear that screen

Enter the following program into your computer and run it.

```
10 PRINT "testing"
20 INPUT A$
30 CLS
```

When you run the program, you'll see the word 'testing' appear under the program listing, more or less as you'd expect. However, below it you'll see a question mark. Where did that come from? The question mark is known as an *input prompt.* An input prompt, which appears in a program when the computer comes to the word INPUT, means the computer is waiting for you to enter something else into the machine, or just to press RETURN. You'll recall that we spoke earlier about strings, and about how they were anything which was enclosed within quote marks. In line 20 above the computer is waiting for a string input (because the A which follows the word INPUT is, in turn, followed by a dollar sign). You can either enter a word, a number, any combination of words, numbers and/or symbols in response to a string input. (But you can only type in a number in response to a numerical input. If you just press the RETURN key when the computer wants a number, the computer will assume you want zero.)

Anyway, when you respond to the input prompt by pressing RETURN, you'll see the screen clears and the word testing disappears. Where did it go? We pointed out that the computer works through a program in line order. Firstly the program printed testing on the screen with line 1Ø and then progressed to line 2Ø, where it waited for an input (or for you to press RETURN). Once you've done this in line 2Ø, the computer moved along to line 3Ø where it found CLS and obeyed that instruction. The instruction was to clear the screen, so the computer did just that and the screen cleared.

Run the program a few times, until you've got a pretty good idea of what is happening, and you've followed through – in your mind – the sequence of steps the computer is executing.

Doing it automatically

Instead of waiting for you to press the RETURN key, you can write a program which clears the screen automatically, as our next example demonstrates. Enter this next program into your computer, type in RUN and then ENTER (or just press the F5, Function 5, key which has exactly the same effect), and then sit back for the Amazing Flashing Word demonstration. Note that you must have spaces on either side of words like FOR and TO.

```
5 CLS:KEY OFF
10 PRINT "autotesting
20 FOR A=1 TO 200
30 NEXT A
40 CLS
50 FOR A=1 TO 200
60 NEXT A
70 GOTO 10
```

Run this program, and you'll see the word autotesting alternatively flashing off and on at the top of the screen. What is happening here? Let's look at the program and go through it line by line. Firstly, as you've probably guessed, line 5 uses CLS to clear the screen and then, after the colon (:) obeys the KEY OFF command to turn off the words across the bottom of the screen. Then, your MSX machine gets to line 1Ø, which prints the word autotesting at the top of the

screen. Next, the computer comes to line 2Ø, where it meets the word FOR. We'll be learning about FOR/NEXT loops (as they are called) in detail in a later chapter, but all you need to know here is that the computer uses FOR/NEXT loops for counting. In this program, lines 2Ø and 3Ø (the FOR is in line 2Ø, the NEXT in line 3Ø) tell the computer to count from one to 5ØØ before moving on.

To stop the program, press Ctrl and the STOP key.

So, it waits for a moment while counting from one to 5ØØ. Then it comes to line 4Ø, which is the command CLS, which tells the computer to clear the screen. The computer then encounters, in lines 5Ø and 6Ø, another FOR/NEXT loop, so waits a while as it counts from one to 5ØØ again. Continuing on in sequence, it comes to 7Ø where it finds the instruction GOTO 1Ø. This, as is immediately obvious, tells the computer to go to line number 1Ø. When the computer gets to line 7Ø, it obeys the GOTO instruction, and starts over from line 1Ø, going through the autotesting printing, counting to 5ØØ, clearing the screen, counting to 5ØØ again, and then coming to GOTO 1Ø so that it starts all over again.

Changing program lines easily

Your computer is provided with an EDIT function which makes it very simple to change the contents of lines within programs. Get rid of the current program by typing the word NEW and then pressing the RETURN key. Then enter the following program into your computer. DO NOT RUN THE PROGRAM. I want to explain something about it before you do.

```
10 REM an edit test
20 PRINT "test again"
30 PRINT "and again"
```

If you want to alter lines which are on the screen, all you have to do is use the arrow keys to move the cursor to the required position, then make the desired changes.

You just move the *cursor* (that 'blob' which follows you around on the screen is the cursor) to the action of line which you want to

change. It is a little different when you want to change lines which are not currently on the screen. You have to bring them into sight so you can run the cursor over them. If you wanted to change, say, line 10, all you'd need to do would be to enter LIST 10, followed by the RETURN key, and line 10 would be reprinted below the rest of the listing.

If you wanted to add an extra word between 'an' and 'test', you would move the cursor across till it was in the space between the words, and then press the key marked with the letters INS (for Insert) before you type in the new word. Try it now, adding the word 'exciting' before the word 'edit', so that your line looks like this:

```
10 REM an exciting edit test
```

Then, press the RETURN key again, then type in LIST (or press function key 4, which has the same effect as typing in LIST) and press RETURN again. This time, when the program is listed, you'll see the new version of line 10 is included within the program.

That was simple, wasn't it. It is just as easy to delete a word, or letter, as it is to add one. Type in LIST 20, and once the line appears on the screen, use the arrow keys to position the cursor where you want it. Now press the DEL (for delete) key, to erase the letters you wish to remove.

If you only have a letter or two wrong within a line, you just move the cursor to the error and then type in the correct letter or letters. These will automatically replace the incorrect material.

Now, these instructions may seem a little complex, and they certainly do not need to be mastered before you can continue your learning. If you're not sure how a particular line should be edited, and you can't be bothered looking it up in your instruction manual, or in this book, just type the whole line again, and when you press RETURN the new line will automatically take the place of the old one within the listing.

Getting the program back

If you want to see a complete listing after it has vanished once a program has been run, all you need to do (as we mentioned briefly a little earlier) is to type in the word LIST, then press the RETURN key.

Then, the whole program listing will appear on the screen. Another way of doing this is to press the key marked 'F4'. This stands for 'function four'. The function keys are preprogrammed on the MSX machines to make it simple to access often-used commands. The first five are the most useful. These are:

1 - COLOR
2 - AUTO
3 - GOTO
4 - LIST
5 - RUN

It is pretty obvious what these stand for and they can save a little time when you're programming on your computer. I suggest you copy out the names of the five, with their corresponding numbers, on a little strip of paper, and place this strip above the keys. Then, you'll know without hesitation what they stand for. You'll probably find, as I have, that a list placed on the computer as I've suggested is much simpler to read than are the words at the bottom of the screen.

If you hold down the SHIFT key, some of the function keys have other functions. These are:

6 - COLOR
7 - CLOAD"
8 - CONT
9 - LIST.
10 - RUN

If you want to define a function key to your own purposes, as I often do when programming, you just enter a command like the following:

```
key 1,"CLS:LIST"+CHR$(13)
```

This will assign function key one to automatically clear the screen and list your program. The + CHR$(13) is the way to program a press of the RETURN key into the function key definition. If you didn't include this, you'd have to press function key one and then press the RETURN key to get it to carry out the commands you'd programmed into the key. Function keys automatically revert to their old functions when you turn the computer off then on again, so you'll need to redefine the keys you want each time you turn the MSX computer on.

There is no reason, when using list, why you must list from the top of the program. When you have longer programs, you may well want to list only part of them. You do this by use of the hyphen (-), as follows:

`LIST - 100`	This lists up to, and including line 100
`LIST 50 - 90`	This lists lines 50 to 90
`LIST 150 -`	This lists the program from line 150 to the end
`LIST 270`	This lists just line 270

Using the printer

Full instructions on printer use come, of course, with the printer, but if you prefer not to bother with them at the moment, and you just want your printer to work, these are the commands you'll need:

LLIST – to list the current program
LPRINT – to print something directly on the printer
LPRINT CHR$(27); "L040" – to set an MSX printer to print to a width of 40 characters.

LLIST is easy to remember, as it is very similar to LIST and has a similar function, except that it lists to the printer rather than listing to the screen. LLIST can be used in the same way as LIST to get just parts of a listing (so LLIST 40 – 70 is valid).

CHAPTER THREE

Descent into Chaos

It's time now to start developing some real programs. You'll notice that from this point on in the book there are some rather lengthy programs. Many of them will contain words from the BASIC programming language which have not been explained. This is because, as programs become more complex (and far more satisfying to run) it becomes more and more difficult to keep words which have not been explained out of the programs. However, this is not a major problem, and you'll probably be able to work out what many of them mean, just from seeing them in the context of a program line.

We are working methodically through the commands available on the computer, and in due course, all of the important ones will be covered. When you come across a word in a program which seems unfamiliar, just type it in. You'll find that you'll soon start picking up the meaning of words which have not been explained, just by seeing how they are used within the program. So if you find a new word, don't worry. The program will work perfectly without you knowing what the word is, and investigating the listing after you've seen the program running is likely to lead you to work out what it means.

Random events

In the world of nature, as opposed to the manufactured world of man, randomness appears to be at the heart of many events. The number of birds visible in the sky at any one time, the fact that it rained yesterday and may rain again today, the number of trees growing on one side of a particular mountain, all appear to be somewhat random. Of course, we can predict with some degree of

certainty whether or not it will rain, but the success of our predictions appears to be somewhat random as well.

When you toss a coin in the air, whether it lands head or tails depends on chance. The same holds true when you throw a six-sided die down onto a table. Whether it lands with the one, the three or the six showing depends on random factors.

Your computer's ability to generate random numbers is very useful in order to get the computer to imitate the random events of the real world. The BASIC word RND lies at the heart of using this means of generating random numbers.

Generating random numbers

We'll start by using RND just as it is to create some random numbers. Enter the following program, and run it for a while:

```
10 PRINT RND(1)
20 GOTO 10
```

When you do, you'll see a list of numbers like these appear on the screen:

```
.59521943994623
.10658628050158
.76597651772823
.57756392935958
.73474759503023
.18426812909758
.37075377905223
.94954151651558
.63799556899423
.47041117641358
```

As you can see, RND generates numbers randomly between zero and one. If you leave it running, it will go on and on apparently forever, writing up new random numbers on the screen.

Now random numbers between zero and one are of limited interest if

we want to generate the numbers and get them to stand for something else. For example, if we could generate 1's and 2's randomly, we could call the 1's heads and the 2's tails and use the computer as a kind of 'electronic coin'. If we could get it to produce whole numbers between one and six, we could use the computer as an imitation six-sided die.

Fortunately, there is a way to do this. Enter the next program and run it:

```
10 PRINT INT(RND(1)*6)+1;
20 GOTO 10
```

When you run this program, you'll get a series of numbers, chosen at random between 1 and 6, like these:

```
 4  1  5  4  5  2  3  6  4  3  5  3  2
1  3  5  1  3  4  3  5  5  4  1  2  3  2
   2  5  1
```

Even though we could create vast series of numbers between 1 and 2 with a program like this, it is not particularly interesting. And, if you ran the program over and over again, you'd find that the sequence of numbers was starting to look very familiar. The random numbers, as you'd discover if you ran the program a number of times, are not really random at all.

This is because the computer does not really generate random numbers, but only looks as if it is doing so. Inside its electronic head, your computer holds a long, long list of numbers, which it prints in order when asked for random numbers. The list is so long, that it is impossible to see a pattern in it, once it is running. However, the series always appears to start at the same point. And in most programs, this is not good enough.

Seeding the random number generator

Fortunately for us, there is a way to get an MSX computer to choose a different spot within the list of numbers each time you run the program, so that the numbers it generates are more nearly random.

Inside your computer is a little value called TIME which is changing all the time your computer is turned on. It would be impossible to predict its value at any point, as it is counting up in ones all the time the machine is on. To make the numbers generated by RND more nearly random, we 'call on' TIME using a line like line 5 in the next program:

```
5 X=RND(-TIME)
10 PRINT INT(RND(1)*6)+1;
20 GOTO 10
```

This chooses a new starting point in the long list of random numbers every time we use it. In some programs in this book you'll see it as it appears in 5 above, and in others the X has been replaced with the word SEED. This is to remind you that the purpose of the line is to 'seed' the random number generator (an obscure computer term which we're stuck with, which simply means to get the random number generator a value to work with).

Putting on the Squeeze

It's time now for our first real program (hooray!). This program shows random numbers in action, and allows you to practise your skills of ESP and prediction.

As the program explains when you run it on your MSX machine, the computer will produce two numbers between one and 13. It will ask you to bet on the probability of the next number it thinks of being between the first two. It's simple to play, and a lot of fun (especially as the money you lose does not really disappear). Don't worry about any commands we haven't discussed yet. Just type the program in as it is and run it. Later on, when you've worked your way right through the book, you can come back to it and you'll be pleased to see you recognise all the commands and functions we've used.

```
10 REM SQUEEZE
20 GOSUB 330
30 GOSUB 80
40 IF D<1 THEN 430
50 GOSUB 350
```

```
60 GOTO 30
70 :
80 PRINT:PRINT:PRINT
90 E=0
100 PRINT "MY FIRST NUMBER IS"A
110 PRINT TAB(6);"MY SECOND IS"B
120 PRINT
130 PRINT "YOU HAVE $"D
140 PRINT
150 PRINT "HOW MUCH DO YOU BET MY NEX
T NUMBER"
160 PRINT "LIES BETWEEN"A"AND"B;
170 INPUT E
180 IF E>D OR E=0 THEN 170
190 D=D-E
200 GOSUB 520
210 PRINT:PRINT "MY NUMBER WAS"C
220 GOSUB 520
230 IF NOT(C>A AND C<B OR C<A AND C>B
) THEN 290
240 PRINT "WELL DONE, YOU WIN $"2*E
250 D=D+3*E
260 GOSUB 520
270 RETURN
280 :
290 PRINT:PRINT "SORRY, YOU LOSE $"E
300 GOSUB 520
310 RETURN
320 :
330 D=20
340 X=RND(-TIME)
350 CLS
360 A=INT(RND(1)*13)+1
370 B=INT(RND(1)*13)+1
380 IF ABS(A-B)<2 OR ABS(A-B)>6 THEN
370
390 C=INT(RND(1)*13)+1
400 IF A=C OR B=C THEN 390
410 RETURN
420 :
430 PRINT
```

```
440 PRINT "THE GAME IS OVER"
450 PRINT
460 PRINT "YOU ARE BROKE!"
470 PRINT
480 PRINT "THANKS FOR THE GAME"
490 GOSUB 520:GOSUB 520
500 END
510 :
520 SOUND 7,49
530 SOUND 6,31
540 SOUND 12,40
550 SOUND 9,7
560 SOUND 10,7
570 SOUND 8,10
580 FOR X=31 TO 0 STEP -1
590 SOUND 2,50-X
600 SOUND 4,60-X
610 SOUND 6,X
620 NEXT X
630 FOR Z=1 TO 200:NEXT Z
640 SOUND 8,0
650 SOUND 9,0
660 SOUND 10,0
670 RETURN
```

Note that some lines (like 70 and 280) simply include a colon(:). This is in the program simply to act as a visual breaker, dividing the program up into sections. Later on, when you write your own programs, you'll appreciate how valuable it is to be able to break up programs in this way so you can see what each part of the program does. It makes it very simple to work out the purpose of each section of the program and – when you're writing your own games – you'll see it also makes it relatively easy to track down errors in the listing.

CHAPTER FOUR

Round and Round We Go

In this chapter, we'll be introducing a very useful part of your programming vocabulary – FOR/NEXT loops. You'll recall that we mentioned FOR/NEXT loops when demonstrating the use of CLS to clear the screen. A FOR/NEXT loop was also used in our squeeze program (line 63Ø) to add a delay.

A FOR/NEXT loop is pretty simple. It takes the form of two lines in the program, the first of which is like this:

```
10 FOR A=1 TO 20
```

With the second like this:

```
20 NEXT A
```

The control variable, the letter after FOR and NEXT, must be the same. (You can, in fact, leave the second A out altogether, as the computer will know what you mean. However, leaving the control variable out makes programs harder to read and alter, so this practice is not recommended in your early programming days.)

As a FOR/NEXT loop runs, the computer counts from the first number up to the second, as these two examples will show:

```
10 FOR A=1 TO 20
20 PRINT A;
30 NEXT A
```

When you run it, you'll see the numbers one to 2Ø appear on the screen, much as you may have expected.

Now try this version:

```
10 FOR A=765 TO 781
20 PRINT A;
30 NEXT A
```

This is the result of running it:

```
 765  766  767  768  769  770  771  7
72  773  774  775  776  777  778  779
  780  781
```

Stepping out

In the two previous examples, the computer has counted up in ones, but there is no reason why it should always count in this way. The word STEP can be used after the FOR part of the first line as follows:

```
10 FOR A=10 TO 100 STEP 10
20 PRINT A;
30 NEXT A
```

When you run this program, you'll discover it counts (probably as you expected) in steps of 10, producing this result:

```
 10  20  30  40  50  60  70  80  90
100
```

Stepping down

The STEP does not have to be positive. Your computer is just as happy counting backwards, using a negative STEP size:

```
10 FOR A=100 TO 10 STEP -10
20 PRINT A;
30 NEXT A
```

This is what the program output looks like:

```
100   90   80   70   60   50   40   30   20
10
```

Making a nest

It is possible to place one or more FOR/NEXT loops within each other. This is called nesting loops. In the next example, the B loop is nested within the A loop:

```
10 FOR A=1 TO 3
20 FOR B=1 TO 2
30 PRINT A"TIMES"B"IS"A*B
40 NEXT B
50 NEXT A
```

The nested program produces this result:

```
1 TIMES 1 IS 1
1 TIMES 2 IS 2
2 TIMES 1 IS 2
2 TIMES 2 IS 4
3 TIMES 1 IS 3
3 TIMES 2 IS 6
```

You need to be very careful to ensure that the first loop started is the last loop which is finished. That is, if FOR A... was the first loop you mentioned in the program, the last NEXT must be NEXT A.

Try swapping line 10 with line 20 in the program, and see what happens when you get your FORs and NEXTs mixed up.

You may recall I said that you do not, in fact, have to mention the control variable with the NEXT if you do not want to. I also said that it was not good programming practice to leave it out as it made programs somewhat difficult to unravel. However, as I imagine you've realised by now, leaving off the control variables at least gets

around the problem of wrongly specifying the NEXT in nested loops.

You can replace lines 40 and 50 of the program with either of the following (removing the old line 50 completely):

```
40 NEXT B:NEXT A
        or
40 NEXT:NEXT
        or
40 NEXT B,A
```

Multiplication tables

You can use nested loops to get the computer to print out the multiplication tables, from one times one right up to twelve times twelve, like this (note that you can omit the semicolon between the parts of the PRINT statement within quote marks and those parts outside; this makes for quicker program entry, but diminishes the readability of the program):

```
10 FOR A=1 TO 12
20 FOR B=1 TO 12
30 PRINT A"TIMES"B"IS"A*B
40 NEXT B
50 NEXT A
```

Here's part of the output:

```
7 TIMES 7 IS 49
7 TIMES 8 IS 56
7 TIMES 9 IS 63
7 TIMES 10 IS 70
7 TIMES 11 IS 77
7 TIMES 12 IS 84
8 TIMES 1 IS 8
8 TIMES 2 IS 16
8 TIMES 3 IS 24
8 TIMES 4 IS 32
8 TIMES 5 IS 40
```

There is no reason why both loops should be travelling in the same direction (that is, why both should be counting upwards) as this variation on the Times Table program demonstrates:

```
10 FOR A=1 TO 12
20 FOR B=12 TO 1 STEP -1
30 PRINT A"TIMES"B"IS"A*B
40 NEXT B
50 NEXT A
```

Here's part of the output of that program:

```
4 TIMES 4 IS 16
4 TIMES 3 IS 12
4 TIMES 2 IS 8
4 TIMES 1 IS 4
5 TIMES 12 IS 60
5 TIMES 11 IS 55
5 TIMES 10 IS 50
5 TIMES 9 IS 45
```

Cracking the code

It's time now for our second games program. In this game which uses several FOR/NEXT loops, CODEBREAKER, the computer thinks of a four-digit number (like 5462) and you have eight guesses in which to work out what the code is. In CODEBREAKER, based on a program by Adam Bennett and Tim Summers, you not only have to work out the four numbers the computer has chosen, but also determine the order they are in.

After each guess, the computer will tell you how near you are to the final solution. A 'white' is the right digit in the wrong position and a 'black' is a correct digit in the right position within the four digits of the code. As you can see from this, you are aiming to get four blacks. Digits may be repeated within the four-number code.

Enter the program and play a few rounds against the computer. Then, return to the book for a discussion on it, which will highlight the role played by the FOR/NEXT loops.

```
10 REM Codebreaker
20 COLOR 11,1
30 CLS
40 X=RND(-TIME)
50 PRINT "           ***************"
60 PRINT "           * CODEBREAKER *"
70 PRINT "           ***************"
80 PRINT
90 PRINT "       When you are told to
do"
100 PRINT "      so, enter a 4-digit n
umber"
110 PRINT "        and then press RETU
RN"
120 PRINT
130 PRINT "        Digits can be repea
ted."
140 PRINT
150 PRINT "       You have 8 goes to b
reak"
160 PRINT "          the difficult cod
e."
170 PRINT
180 PRINT "           ***************"
190 FOR W=1 TO 4:GOSUB 890:GOSUB 970
200 NEXT W
210 CLS
220 DIM B(4),D(4)
230 H=0
240 FOR A=1 TO 4
250 B(A)=INT(RND(1)*9)+1
260 NEXT A
270 :
280 FOR C=1 TO 8
290 PRINT
300 IF RND(1)<.5 THEN COLOR 3,1 ELSE
COLOR 5,1
310 BEEP:FOR Z=1 TO 90:NEXT Z:BEEP
320 PRINT TAB(7);"Enter guess number"
C
330 INPUT "                  ";X
```

```
340 IF X>9999 THEN 320
350 IF X<1000 THEN 320
360 :
370 P=INT(X/1000)
380 Q=INT((X-1000*P)/100)
390 R=INT((X-1000*P-100*Q)/10)
400 S=INT(X-1000*P-100*Q-10*R)
410 D(1)=P
420 D(2)=Q
430 D(3)=R
440 D(4)=S
450 PRINT TAB(11);
460 FOR E=1 TO 4
470 IF D(E)<>B(E) THEN 530
480 PRINT "Black ";
490 GOSUB 890
500 B(E)=B(E)+10
510 D(E)=D(E)+20
520 H=H+1
530 NEXT E
540 IF H=4 THEN 810
550 :
560 FOR F=1 TO 4
570 D=D(F)
580 FOR G=1 TO 4
590 IF D<>B(G) THEN 640
600 PRINT "White ";
610 GOSUB 970
620 B(G)=B(G)+10
630 GOTO 650
640 NEXT G
650 NEXT F
660 :
670 FOR G=1 TO 4
680 IF B(G)<10 THEN 700
690 B(G)=B(G)-10
700 NEXT G
710 :
720 H=0
730 PRINT
740 NEXT C
```

```
750 :
760 PRINT:PRINT "You didn't get it...
"
770 PRINT
780 PRINT "The answer is:  ";B(1);B(2
);B(3);B(4)
790 GOTO 850
800 :
810 PRINT:PRINT:PRINT "Well done, cod
ebreaker!"
820 PRINT
830 PRINT:PRINT "You got the answer i
n"
840 PRINT TAB(5);"just"C"goes"
850 PRINT
860 PRINT
870 END
880 :
890 SOUND 0,1:SOUND 8,5
900 FOR Z=10 TO 1 STEP -1.5
910 SOUND 1,Z
920 FOR Q=1 TO 50:NEXT Q
930 NEXT Z
940 SOUND 8,0
950 RETURN
960 :
970 SOUND 0,1:SOUND 8,5
980 FOR Z=20 TO 30
990 SOUND 1,Z
1000 FOR Q=1 TO 50:NEXT Q
1010 NEXT Z
1020 SOUND 8,0
1030 RETURN
```

And here is the end of one round played against the program:

Black White

Enter guess number 3 ? 9854
White White

Enter guess number 4 ? 3243

```
Enter guess number 5 ? 7854
  Black  White

Enter guess number 6 ? 6547
  White  White

Enter guess number 7 ? 8976
  Black  Black  White  White

Enter guess number 8 ? 8967
  Black  White  White  White

You didn't get it . . .
The answer is:  7  9  8  6
```

We'll now go through the program, line by line, a practice we'll be following in some of the other programs in this book. If you don't want to read the detailed explanation now (and there may well be parts of it which are a bit difficult to understand at your present stage), by all means skip over the explanation and then come back to it later when you know a little more.

Lines 5Ø, 7Ø and 18Ø print a number of asterisks to rule off the title and instructions, with blank lines printed by 8Ø, 12Ø, 14Ø and 18Ø. The random number generator is seeded, as we discussed earlier, in line 4Ø.

Line 19Ø creates a pause for a few seconds so that you can read the instructions, before the screen is cleared by line 21Ø. Arrays are dimensioned in line 22Ø. We discuss arrays in a later chapter. For now, all you need to know is that by saying DIM B(4) you tell the computer you want to create a list of objects, with the list called B, in which the first item can be referred to as B(1), the second as B(2) and so on. You do not really need to dimension an array when less than 11 elements will be needed, but it helps to keep your thinking clear to always dimension arrays before using them in programs. In this program the arrays are used for storing the numbers picked by the computer, and for storing the digits which you pick each time you try to break the code.

H is a numeric variable (we've mentioned numeric variables before, you'll recall) which is set equal to zero in line 23Ø. In line 52Ø, one is

added to the value of H each time a black is found, so that if H ever gets to equal four, the computer knows all the digits have been guessed, and goes to the routine from line 81Ø to print up the congratulations.

The lines from 24Ø to 26Ø work out the number which you will have to try and guess. Line 25Ø uses the RND function we've discussed before to get four random numbers between zero and nine, and stores one each in the elements of the B array. Note that the first FOR/NEXT loop of our program appears here. The A in line 25Ø equals one the first time the loop is passed through, two the second time, and so on.

Our next FOR/NEXT loop, which uses C, starts in the next line. It counts from one to eight, to give you eight guesses. Line 33Ø accepts your guess, after the previous line has told you which guess it is you are entering. The numeric variable X is set equal to your guess, and checks are made to make sure you have not entered a five-digit number (line 34Ø) or one which has less than four digits (line 35Ø). If you have, the program goes back to line 32Ø to ask you once again to enter a guess.

The next section of the program, right through to line 7ØØ, works out how well you've done, using a number of FOR/NEXT loops (46Ø to 53Ø, 56Ø to 65Ø, 58Ø to 64Ø and 67Ø to 7ØØ). Line 74Ø sends the program back to the line after the FOR C =... to go through the loop again. If the C loop has been run through eight times, then the program does not go back to line 29Ø, but 'falls through' line 74Ø to 76Ø to tell you that you have not guessed the code in time, and to tell you what it is. Line 78Ø prints out the code.

If you do manage to guess it, so that H equals four in line 54Ø, then the program jumps to line 81Ø to print out the congratulatory message.

CHAPTER FIVE

Changing in Mid-stream

We pointed out at the beginning of the book that, in most situations, your computer follows through a program in line order, starting at the lowest line number and following through in order until the program reaches the final line.

This is not always true. The GOTO command sends action through a program in any order which you determine. Enter the following program, and before you run it, see if you can predict what the result of running it will be:

```
10 GOTO 40
20 PRINT "THIS IS 20"
30 GOTO 60
40 PRINT "THIS IS 40"
50 GOTO 20
60 PRINT "THIS IS 60"
70 FOR Z=1 TO 200:NEXT Z
80 GOTO 40
```

This rather pointless program sends the poor MSX computer jumping all over the place, changing its position in the program every time it comes to a GOTO command. Here's what you should have seen on your screen:

```
THIS IS 40
THIS IS 20
THIS IS 60
THIS IS 40
THIS IS 20
THIS IS 60
```

```
THIS IS 40
THIS IS 20
THIS IS 60
THIS IS 40
THIS IS 20
THIS IS 60
```

The program starts at line 1Ø, and finding GOTO 4Ø there, moves onto line 4Ø to print the message "THIS IS 40". It then continues on to line 5Ø where it finds the instruction GOTO 2Ø. Without question, it zips back to line 2Ø to print out "this is 2Ø" then goes to line 3Ø which directs it to line 6Ø. At line 6Ø it finds the instruction to print out "this is 6Ø" which it obeys. The computer then follows through to line 7Ø where the Z loop inserts a short pause, before the computer moves on to line 8Ø to find yet another GOTO instruction, this time to line 4Ø, which is just about where we began ... and the whole thing starts over again.

Restrictive practices

Using GOTO in this way is called unconditional branching. The command is not qualified in any way, so the computer always obeys it. This brings us neatly to the next computer words we will consider. These are a pair of words IF and THEN, nearly always found (or implied) together, which impose conditions on branching by GOTO commands. This pair of words is easy to understand. IF something is true, THEN do something else. IF you are hungry, THEN order a hamburger. IF you want a big car, THEN save for it. IF something THEN something.

The next program, which 'rolls a die' (using the random number generator) and then prints up the result of that die roll as a word, uses a number of IF/THEN lines:

```
10 REM DICE ROLLS
20 GOTO 140
30 PRINT "ONE"
40 GOTO 140
50 PRINT "TWO"
60 GOTO 140
70 PRINT "THREE"
80 GOTO 140
90 PRINT "FOUR"
100 GOTO 140
110 PRINT "FIVE"
120 GOTO 140
130 PRINT "SIX"
140 A=INT(RND(1)*6)+1
```

```
150 FOR Z=1 TO 200:NEXT Z
160 IF A=1 THEN GOTO 30
170 IF A=2 THEN GOTO 50
180 IF A=3 THEN GOTO 70
190 IF A=4 THEN GOTO 90
200 IF A=5 THEN GOTO 110
210 IF A=6 THEN GOTO 130
```

This is what you'll see when you run the program:

```
FOUR
ONE
FIVE
FOUR
FIVE
TWO
THREE
SIX
FOUR
THREE
FIVE
```

So, we've looked at non-conditional and conditional GOTO's to send action all over the place within a program.

Subroutines, another way to fly

There is another way to redirect the computer during the course of a program. This is by the use of subroutines. A subroutine is part of a program which is run twice or more during a program, and is more efficiently kept outside the main program than within it.

The next program should make it clear. In this, the computer throws a die over and over again. The first time it is thrown, the computer is throwing it for you. The second time it throws the die for itself. After each pair of dice has been thrown, it will announce who is the winner (highest number wins). The program uses a subroutine to throw the die, so we do not need two identical 'die-throwing routines' within a single program. Enter and run the program, then return to the book, and I'll explain where the subroutine is within the program, and how it works:

```
10 FOR Z=1 TO 500:NEXT Z
20 PRINT:PRINT
30 FOR C=1 TO 2
40 GOSUB 130
50 IF C=1 THEN A=D
60 IF C=2 THEN B=D
70 NEXT C:PRINT
80 IF A>B THEN PRINT TAB(9);"I win"
90 IF A<B THEN PRINT TAB(9);"You win"
100 IF A=B THEN PRINT TAB(9);"It's a
draw"
110 BEEP
120 GOTO 10
130 REM :: This is the subroutine ::
140 D=INT(RND(1)*6)+1
150 IF C=1 THEN PRINT:PRINT "I rolled
 a"D
160 IF C=2 THEN PRINT:PRINT "You roll
ed a"D
170 FOR Z=1 TO 100:NEXT Z:X=RND(-TIME
)
180 RETURN
```

This is what you'll see when you run it:

```
I rolled a 2

You rolled a 3

          You win
I rolled a 6

You rolled a 3

          I win
I rolled a 5

You rolled a 5

          It's a draw
```

The program pauses for a short while on line 1Ø, prints two blank lines, then enters the C FOR/NEXT loop. When it gets to line 4Ø which it does (of course) once each time through the C loop, the program is sent to the subroutine starting at line 14Ø. The 'die is rolled' in line 14Ø, and the numeric variable D is set equal to the result of the roll. The next two lines print out the result of the roll, using an IF/THEN to determine whether the computer should print "I ROLLED A ..." or "YOU ROLLED A ...". There is a slight pause (line 17Ø) and then the computer comes to the word RETURN. The word RETURN signals to the computer that it must return to the line after the one which sent it to the subroutine. In this program, that line (the one which is after the one which sent it to the subroutine) is 50.

There, the IF/THENs in lines 5Ø and 6Ø determine whether the value of the roll (D) should be assigned to the variable A or to B.

Line 7Ø ends the FOR/NEXT loop, and then lines 8Ø to 1ØØ determine whether the computer has won (which it will have done if A is greater than B, a condition which is tested using the > sign in line 8Ø) or whether the human has won (which will happen if A is less than B, a condition tested in line 9Ø by the 'less than' symbol, <). From here the program goes back to line 1Ø where it starts again. (By the way, you get the computer to stop running an 'endless' program of this type by holding down the CTRL key and then pressing the STOP key until the program halts.)

Study this program, until you're pretty sure you know how subroutines work.

Let's roll again

You may wonder if it is possible to change the earlier program, which changed the number rolled by the die into a word, using subroutines. The answer is 'yes', although the program with subroutines seems, at first sight, not much stronger than the GOTO version, and certainly it is not any clearer. Here's one way it could be done:

```
10 REM Dice Rolls
20 GOTO 140
30 PRINT "ONE"
40 RETURN
50 PRINT "TWO"
60 RETURN
70 PRINT "THREE"
80 RETURN
90 PRINT "FOUR"
100 RETURN
110 PRINT "FIVE"
120 RETURN
130 PRINT "SIX"
140 A=INT(RND(1)*6)+1
150 IF A=1 THEN GOSUB 30
160 IF A=2 THEN GOSUB 50
170 IF A=3 THEN GOSUB 70
180 IF A=4 THEN GOSUB 90
190 IF A=5 THEN GOSUB 110
200 IF A=6 THEN GOSUB 130
210 FOR Z=1 TO 200:NEXT Z
220 GOTO 140
```

ON GOSUB

However, there is a way to do it cleanly, using ON... GOSUB. This means that the computer can choose from a number of subroutine destinations, depending on the value which has been assigned to a variable.

To try and make that clear, here is another version of the dice roll program, using ON... GOSUB:

```
10 REM ON...GOSUB ROLLS
20 FOR Z=1 TO 200:NEXT Z
30 A=INT(RND(1)*6)+1
40 ON A GOSUB 60,80,100,120,140,160
50 GOTO 20
60 PRINT "ONE"
70 RETURN
```

```
80 PRINT "TWO"
90 RETURN
100 PRINT "THREE"
110 RETURN
120 PRINT "FOUR"
130 RETURN
140 PRINT "FIVE"
150 RETURN
160 PRINT "SIX"
170 RETURN
```

Look first at line 30. This assigns a value, chosen randomly between one and six, to the variable A. You may have expected the line to read LET A= ... and so on. However, the LET is optional. It often makes the meaning of the line clear, so you can leave it in if you like. But, as you'll see when you run the program, it makes no difference to the computer.

Now the most important statement in the program, line 40. This means that if A equals 1, GOSUB the first number to follow the GOSUB command. If A equals 2, go to the second number; if A equals three go to the third, and so on.

The program can be further condensed by the use of colons. Colons allow you to place more than one program statement after a single line number. When the RETURNS are placed on the same line as the PRINT statements, as in the following version, the program closes up even more:

```
10 REM ON...GOSUB ROLLS
20 FOR Z=1 TO 200:NEXT Z
30 A=INT(RND(1)*6)+1
40 ON A GOSUB 60,70,80,90,100,110
50 GOTO 20
60 PRINT "ONE":RETURN
70 PRINT "TWO":RETURN
80 PRINT "THREE":RETURN
90 PRINT "FOUR":RETURN
100 PRINT "FIVE":RETURN
110 PRINT "SIX":RETURN
```

Note also that the lines have been renumbered, so they are all in neat 1Ø's. You can do this very simply, just by typing in RENUM and then pressing RETURN.

We all know the equals sign (=) and we've seen it in use in several programs before. We've also seen the 'greater than' (>), the 'less than' (<) and the 'not equals to' (<>). At this point of the book, I thought it would be useful to briefly recap on what each of these signs are, and what they mean:

= equals
> greater than
< less than
>= greater than or equal to
<= less than or equal to
<> not equal to

The words AND and OR can also be used in comparison lines, chaining tests together. These two words work as follows:

AND The computer does what follows the THEN if both of the conditions chained by the AND are true

OR The computer carries out the instruction following the THEN if either of the conditions is true

CHAPTER SIX

Stringing Along

You'll recall that several times in this book so far we have referred to numeric variables (letters like A or B, words like COUNT and GUESS, and combinations such as R2D2 and C3PØ) and to string variables (one or more letters followed by a dollar sign, such as NAME$, A$ or AGE$ is a string variable). In this chapter, we'll be looking at strings, and at things you can do with them.

The character set

Every letter, number or symbol the MSX computer prints has a code (the code, by the way, is an ASCII code and ASCII is explained in the glossary). Telling the computer to print the character of that code produces the character.

It is easy to understand this. As the code is an ASCII code, as I pointed out above, the computer word for the code is ASC. Note that the ASC value for the letter "A" has nothing to do with the value assigned to A when it is a numeric variable, but refers to "A" when we actually want the computer to print the letter "A". Note that we put the "A" in quote marks when we're referring to it as a letter.

Try it now. Enter the following into your computer, and see what you get:

```
PRINT ASC("A")
```

Note that the letter for which you want the ASC must be within parentheses and also within quote marks, as above. Now when you get the computer to run the above line, it should give the answer 65.

(If you didn't get that, you either missed something out in the line or you're using an "a" instead of an "A".)

From this we can see that 65 is the ASC (ASCII code) of "A". We can turn a 65 back into an "A" by asking the computer to print the character which corresponds to ASC code 65. We do this with the BASIC word CHR$, as follows:

```
PRINT CHR$(65)
```

Run this, and the letter "A" will appear. You can get your MSX computer to print out every ASC code and its character with the next short program. Enter it, and watch closely:

```
10 REM Showing ASC and CHR$
20 FOR A=32 TO 255
30 PRINT A;CHR$(A);"  ";
40 FOR B=1 TO 50:NEXT B
50 NEXT A
```

This is the printout you'll see:

```
32      33 !    34 "    35 #    36 $    37 %
38 &    39 '    40 (    41 )    42 *    43 +
44 ,    45 -    46 .    47 /    48 0    49 1
50 2    51 3    52 4    53 5    54 6    55 7
56 8    57 9    58 :    59 ;    60 <    61 =
62 >    63 ?    64 @    65 A    66 B    67 C
68 D    69 E    70 F    71 G    72 H    73 I
74 J    75 K    76 L    77 M    78 N    79 O
80 P    81 Q    82 R    83 S    84 T    85 U
86 V    87 W    88 X    89 Y    90 Z    91 [
92 \    93 ]    94 ^    95 _    96 `    97 a
98 b    99 c    100 d
101 e    102 f    103 g    104 h    105 i
106 j    107 k    108 l    109 m    110 n
111 o    112 p    113 q    114 r    115 s
116 t    117 u    118 v    119 w    120 x
121 y    122 z    123 {    124 |    125 }
126 ~    127    128 Ç    129 ü    130 é
```

131	â	132	ä	133	à	134	à	135	ç
136	ê	137	ë	138	è	139	ï	140	î
141	ì	142	Ä	143	Å	144	É	145	æ
146	Æ	147	ô	148	ö	149	ò	150	û
151	ù	152	ÿ	153	Ö	154	Ü	155	¢
156	£	157	¥	158	₧	159	ƒ	160	á
161	í	162	ó	163	ú	164	ñ	165	Ñ
166	ª	167	º	168	¿	169	⌐	170	¬
171	½	172	¼	173	¡	174	«	175	»
176	ã	177	ã	178	Ĩ	179	ĩ	180	õ
181	õ	182	Õ	183	Õ	184	Ij	185	ij
186	¾	187	˜	188	◇	189	⅞	190	¶
191	§	192	▁	193	▚	194	■	195	▔
196	-	197	▀	198	▌	199	▞	200	▐
201	▕	202	▊	203	//	204	\\	205	▼
206	▲	207	▶	208	◀	209	⧗	210	⋈
211	▪	212	▫	213	▪	214	▫	215	⋇
216	Δ	217	∓	218	ω	219	▉	220	▃
221	▋	222	▋	223	▀	224	α	225	ß
226	Γ	227	π	228	Σ	229	σ	230	μ
231	ɣ	232	⁑	233	θ	234	Ω	235	δ
236	∞	237	⌀	238	∈	239	∩	240	≡
241	±	242	≥	243	≤	244	⌠	245	⌡
246	÷	247	≈	248	°	249	∙	250	·
251	√	252	ⁿ	253	²	254	▮	255	

Testing your character

Our next program is a reaction tester. In this program, you have to try and find the *right key* on the keyboard as quickly as possible.

Make sure that CAPS LOCK is engaged when you run the program. A letter will appear on the screen. As quickly as you can, find that letter on the keyboard and press it. You'll be told how long it took you, and this time will be compared with your best time.

Notice how the letter which is printed on the screen uses CHR$ in line 8Ø, printing the character of the number chosen at random by line 5Ø and assigned there to variable A. A$ is set equal to INKEY$ (which is explained a little later in the book) in line 9Ø and compared with the letter the computer has chosen in line 1ØØ.

```
10 REM CHARACTER TEST
20 CLS:KEY OFF
30 X=RND(-TIME)
40 BEST=1000
50 A=65+INT(RND(1)*26)
60 B=0
70 LOCATE 13,7
80 PRINT CHR$(A)
90 A$=INKEY$
100 IF A$=CHR$(A) THEN BEEP:GOTO 180
110 BEEP
120 B=B+3.7
130 LOCATE 9,4
140 PRINT B
150 IF B<400 THEN 90
160 PRINT "Sorry, time is up!"
170 GOTO 220
180 CLS
190 LOCATE 4,4
200 PRINT "Well done, you scored"
210 PRINT:PRINT TAB(6);B"on that one"
220 IF B<BEST THEN BEST=B
230 PRINT:PRINT:PRINT
240 IF BEST<>1000 THEN PRINT CHR$(210
);" The best score so far is"BEST;CHR
$(210)
250 FOR G=1 TO 15*B
260 NEXT G
270 CLS
280 GOTO 50
```

Cutting them up

One of the very useful aspects of the BASIC on your MSX computer is the way it can be used to manipulate strings. The words used to handle strings are:

```
LEFT$
MID$
RIGHT$
```

(By the way, these are usually spoken aloud as 'left-string', 'mid-string' and 'right-string'.)

The next program shows them in action. Enter it and run it on your computer, then return to the book for a discussion to show what can be learned from it.

```
10 CLS
11 KEY OFF
12 A$="FIFTH*AVENUE"
13 PRINT
14 PRINT "LEFT$(A$,3)="LEFT$(A$,3)
15 PRINT
16 PRINT "LEFT$(A$,5)="LEFT$(A$,5)
17 PRINT
18 PRINT "RIGHT$(A$,3)="RIGHT$(A$,3)
19 PRINT
20 PRINT "RIGHT$(A$,5)="RIGHT$(A$,5)
21 PRINT
22 PRINT "MID$(A$,3)="MID$(A$,3)
23 PRINT
24 PRINT "MID$(A$,5)="MID$(A$,5)
25 PRINT
26 PRINT "MID$(A$,5,4)="MID$(A$,5,4)
27 PRINT
28 PRINT "MID$(A$,2,7)="MID$(A$,2,7)
```

As you can see, the program first (in line 20) sets A$ equal to "FIFTH*AVENUE". Then it uses LEFT$, RIGHT$ and MID$ to extract the original string, A$.

Here's what it looks like when you run it:

```
LEFT$(A$,3)=FIF

LEFT$(A$,5)=FIFTH

RIGHT$(A$,3)=NUE

RIGHT$(A$,5)=VENUE

MID$(A$,3)=FTH*AVENUE
```

```
MID$(A$,5)=H*AVENUE

MID$(A$,5,4)=H*AV

MID$(A$,2,7)=IFTH*AV
```

Look at the first line of the output. LEFT$(A$,3) = FIF. LEFT$ takes the *leftmost* portion of the string as far as the number which follows the string. That is, when we have LEFT$(A$,3) it takes the three leftmost characters of the string. The next printout, LEFT$(A$,5) takes the five leftmost characters of the string, producing in this case FIFTH (because they are five leftmost characters of the overall string).

It can be used slightly differently. If we said:

```
PRINT LEFT$("FIFTH*AVENUE",3)
```

the computer would print out FIF. The string, then, can either be a string variable (A$) or the string in full ("FIFTH*AVENUE").

As you've probably worked out by now, RIGHT$ does the same thing as LEFT$, except it starts at the righthand end of the string. Therefore, RIGHT$(A$,3) selects the three rightmost characters of the string, in this case NUE. Again, as above, this is the same as saying:

```
PRINT RIGHT$("FIFTH*AVENUE",3)
```

MID$ is a little more flexible. It selects a portion from the middle of the string, *starting from* the character number which follows the string. Therefore, MID$(A$,4) prints all the string starting with the fourth character.

If there is only one number (such as the 4 above), then MID$ selects *all* of the string to the end of it. However, if there is another number, this second number dictates the *length* of the string which will be extracted.

You can see in the last two printouts from the program that MID$(A$,5,4) prints the extract of the string four characters long,

starting from character five. MID$(A$,2,7) produces a string seven characters long, starting from the second character.

Return the program now, putting your name in place of FIFTH*AVENUE in line 2Ø.

Putting them back together

Strings can be added together on an MSX computer. The process of adding strings is called the frightening-looking word *concatenation.* You can concatenate two or more complete strings together, or just add bits of them, as our next program shows:

```
100 REM CONCATENATION
110 X=RND(-TIME)
120 CLS:KEY OFF
130 A$="AMERICA"
140 B$="COLUMBUS"
150 C$=A$+B$
160 PRINT "A$ = "A$
170 PRINT
180 PRINT "B$ = "B$
190 PRINT
200 PRINT "C$ = "C$
210 PRINT
220 D=INT(RND(1)*6)+1
230 E=INT(RND(1)*6)+7
240 PRINT "MID$(C$"D","E") = "MID$(C$
,D,E)
250 PRINT
260 D$=MID$(C$,D,E)
270 E$=A$+D$
280 PRINT "E$ = "E$
```

When you run this program, which creates C$ in line 1ØØ by concatenating A$ and B$, you'll see results like these two:

```
A$ = AMERICA

B$ = COLUMBUS
```

```
C$ = AMERICACOLUMBUS

MID$(C$ 5 , 12 ) = ICACOLUMBUS

E$ = AMERICAICACOLUMBUS

--------------------------------

A$ = AMERICA

B$ = COLUMBUS

C$ = AMERICACOLUMBUS

MID$(C$ 4 , 12 ) = RICACOLUMBUS

E$ = AMERICARICACOLUMBUS
```

Playing around

You can do a number of things with string manipulation, as our next program demonstrates. NAME PYRAMID allows you to enter your name to produce a very interesting display. Once you've seen the program running, you'll understand why the program has been given the name it has.

This is the listing of NAME PYRAMID:

```
1 REM Name Pyramid
2 CLS:KEY OFF
3 INPUT "What is your full name";A$
4 IF LEN(A$)>15 THEN A$=LEFT$(A$,15)
5 A=LEN(A$)
6 CLS
7 FOR G=1 TO A
8 PRINT TAB(16-G);
9 FOR H=1 TO 2*G
10 PRINT MID$(A$,G,1);
11 NEXT H
12 PRINT
13 NEXT G
```

And here are two runs of the program:

```
TT
iiii
mmmmmm

HHHHHHHHHH
aaaaaaaaaaa
rrrrrrrrrrrrrr
ttttttttttttttt
nnnnnnnnnnnnnnnnnn
eeeeeeeeeeeeeeeeeee
lllllllllllllllllll
lllllllllllllllllllll

MM
SSSS
XXXXXX

CCCCCCCCCC
ooooooooooo
mmmmmmmmmmmmm
ppppppppppppppp
uuuuuuuuuuuuuuuuuuu
ttttttttttttttttttt
.eeeeeeeeeeeeeeeeeeee
rrrrrrrrrrrrrrrrrrrrrrrr
!!!!!!!!!!!!!!!!!!!!!!!!!
```

Playing it back

Our final program in this chapter shows one very effective use of string manipulation, in which a string is progressively reduced by one element.

When you run ECHO GULCH, you'll see a letter appear on the screen. It will then vanish. Once it has vanished, you will have a limited amount of time in which to press the key yourself.

If you've pressed the right key, a beep will sound, and the letter will be replaced with a new one. This will stay on the screen for a shorter time than the previous one.

Each time a new letter appears, you will be given less time to see it, before you have to press that particular key on the keyboard. If you make a mistake, the "SORRY, THAT IS WRONG" message will appear, along with your score. If you manage to get all the list of letters right, you'll be rewarded with a "YOU'RE THE CHAMP!!" message.

Here's the listing:

```
10 REM Echo Gulch
20 REM Maximum score is 30
30 S=0
40 A$="ABSCHDEUQZAFKJHJHEUSCLEKLKDJSK
FH"
50 CLS:KEY OFF
60 LOCATE 12,15
70 PRINT MID$(A$,1,1)
80 FOR G=1 TO 17*LEN(A$):NEXT G
90 CLS
100 B$=INKEY$
110 IF B$=MID$(A$,1,1) THEN BEEP:BEEP
:S=S+1
120 IF B$<"A" OR B$>"Z" THEN 100
130 LOCATE 12,15
140 PRINT B$
150 LOCATE 5,1
160 PRINT "Your score is"S
170 LOCATE 12,15
180 PRINT " ":REM single space
190 IF B$<>MID$(A$,1,1) THEN 240
200 A$=MID$(A$,2)
210 IF LEN(A$)=2 THEN 270
220 FOR G=1 TO 500-10*S:NEXT G
230 GOTO 60
240 PRINT:PRINT "Sorry, that is wrong
"
250 PRINT:PRINT "You scored"S
260 END
270 PRINT "You're the champ!"
```

The variable S, which holds your score, is set to zero in line 3Ø, and line 4Ø sets the string variable A$ to a long line of letters. Line 7Ø prints the first letter only of the string, and line 8Ø inserts a short delay loop, which uses the LEN function.

This is another string function, and returns the length of a string, that is, the number of characters which make it up. LEN does not make any distinction between letters, numbers, symbols or spaces, as you'll discover if you enter a number of PRINT LEN A$ statements, after setting A$ to equal various words, symbols and sentences.

Because, in our program A$ is reduced by one character by line 2ØØ each time the program cycles, LEN A$ is a smaller number each time. Therefore, the delay produced by line 8Ø (which dictates how long the character will be on the screen before it vanishes) becomes shorter.

INKEY$

Line 1ØØ uses INKEY$ to read the keyboard. INKEY$, as you've probably worked out by this point in the book, does not demand that you press RETURN after touching a key. INKEY$ always returns the key you have pressed as a string. INKEY$ does not, in contrast to INPUT, wait until you have pressed a key before the program continues.

If you are not touching a key when the program comes to an INKEY$, it simply passes right through the line, reading your non-touching of the keyboard as the null string (two quote marks with nothing, not even a space, between them, as " ").

Line 12Ø looks at B$, the variable which is set equal to whichever key is being pressed as the program goes through line 1ØØ. As you can see in line 12Ø, you can use the greater than (>) and the less than (<) symbols, which we discussed in chapter five, in connection with strings. These look at all elements of a string and compare them in terms of alphabetical order (so "ZEBRA" is *greater than* "AARDVARK" and "BEAST" is *less than* "BEAUTY").

As well, you can compare strings using equals (=) and not equals (<>) as is shown by the next few lines of the program. Line 11Ø compares the key you have pressed with the first element of A$, and if they are the same, continues through the multistatement line to BEEP and then adds one to your score (variable S).

Line 18Ø then blanks out the letter, in preparation for the next one to appear. Line 19Ø compares B$ with the first element of A$ again, and if it finds they are not equal, sends the program to lines 24Ø and 25Ø where you are told "SORRY THAT IS WRONG" and your score is given.

Line 2ØØ strips the string A$ of its first character, by setting A$ equal to MID$(A$,2). Line 21Ø checks to see if the length of A$ equals 2 (that is, if LEN A$ = 2) and if it finds that it is, goes to line 27Ø to print out the "YOU'RE THE CHAMP!!" message. If not, the program cycles back to line 6Ø to print out the next letter for you.

CHAPTER SEVEN

A Game and a Test

It's time now to take a break from the serious business of learning to program your MSX computer. As you can see in this chapter, we have two major programs which use many commands which have not yet been explained. I suggest you enter the programs just as they are, play them for your own enjoyment, then come back to the explanations which follow the listings after you've mastered the rest of the book.

I do not think it's fair to keep you waiting for major programs until you've covered everything on the computer. Also, entering short demonstration programs can get pretty boring if you're longing to see your computer *really* in action. Therefore, I hope you'll enter the programs 'on trust', returning to this chapter for the explanations when you feel you are ready. Of course, you do not have to enter the programs right now. If you'd prefer to continue with the learning, then move straight along to the next chapter.

The first game in this chapter is *MSX-thello,* a version of the game known as *Reversi* or *Othello* (note that *Othello* is a registered trademark of Gabriel Industries, Inc., USA and of Mine of Information, UK, when used in connection with computer versions). Invented in the late 1800's, *MSX-thello* is played on an eight by eight board. You use pieces which have different colours on each side.

The game begins with four pieces placed on the centre squares, as you'll see when you run the program. You move by placing one of your pieces next to a computer piece or pieces, with another of your pieces further on. When that happens, all the computer pieces "reverse" to become your own pieces.

Here's how it works. Suppose a line of pieces looked like this:

0XXXX

and you decided to put your piece (the Ø) at the end of the line like this:

0XXXXØ

The computer pieces would reverse, so the line looked like this after your move:

000000

The game continues until every square on the board is filled, or neither player can move. Fortunes change swiftly in this game, as rows branching off your position (such as on the diagonals) can be changed with a single move. If you cannot move at any time, you signal this to your computer by entering a zero. You are the filled-in ovals, and the MSX computer is the open ovals.

```
10 REM Mx-thello
20 GOTO 970
30 LOCATE 12,0
40 PRINT "My move..."
50 TIME=0
60 S=OX:T=X:H=0
70 FOR A=2 TO 9
80 FOR B=2 TO 9
90 IF A(A,B)<>46 THEN 280
100 Q=0
110 FOR C=-1 TO 1
120 FOR D=-1 TO 1
130 LOCATE 3,0:PRINT TIME/50
140 K=0:F=A:G=B
150 IF A(F+C,G+D)<>S THEN 180
160 K=K+1:F=F+C:G=G+D
170 GOTO 150
180 IF A(F+C,G+D)<>T THEN 200
190 Q=Q+K
200 NEXT D
```

```
210 NEXT C
220 IF A=2 OR A=9 OR B=2 OR B=9 THEN
Q=Q*2
230 IF A=3 OR A=8 OR B=3 OR B=8 THEN
Q=Q/2
240 IF (A=2 OR A=9) AND (B=3 OR B=8)
OR (A=3 OR A=8) AND (B=2 OR B=9) THEN
 Q=Q/2
250 IF Q<H OR (RND(1)<.3 AND Q=H) THE
N 280
260 LOCATE 3,0:PRINT TIME/50
270 H=Q:M=A:N=B
280 NEXT B
290 NEXT A
300 IF H=0 AND R=0 THEN 890
310 IF H=0 THEN 330
320 GOSUB 740
330 GOSUB 470
340 LOCATE 0,0
350 INPUT "        Enter your move ";R
360 REM -- Enter zero to pass --
370 S=X:T=OX
380 IF R=0 THEN 440
390 IF R<11 OR R>88 THEN 340
400 R=R+11
410 M=R/10
420 N=R-10*M
430 GOSUB 740
440 GOSUB 470
450 GOTO 30
460 :
470 REM ** Print Board **
480 C=0:H=0
490 BEEP
500 LOCATE 0,0
510 PRINT "
      "
520 PRINT:PRINT
530 PRINT TAB(9);"MSX Machine is "CHR
$(X)
540 PRINT TAB(11);"Human is ";CHR$(OX
```

```
)
550 PRINT
560 PRINT TAB(10);"1 2 3 4 5 6 7 8"
570 FOR B=2 TO 9
580 PRINT TAB(6);B-1;" ";
590 FOR D=2 TO 9
600 PRINT CHR$(A(B,D));" ";
610 IF A(B,D)=X THEN C=C+1
620 IF A(B,D)=OX THEN H=H+1
630 NEXT D
640 PRINT B-1
650 NEXT B
660 PRINT TAB(10);"1 2 3 4 5 6 7 8"
670 PRINT:PRINT
680 PRINT TAB(10);"MSX Machine:"C
690 PRINT:PRINT TAB(12);"Human:"H
700 BEEP
710 IF C+H=64 THEN 900
720 RETURN
730 :
740 FOR C=-1 TO 1
750 FOR D=-1 TO 1
760 F=M:G=N
770 IF A(F+C,G+D)<>S THEN 810
780 F=F+C:G=G+D
790 GOTO 770
800 :
810 IF A(F+C,G+D)<>T THEN 850
820 A(F,G)=T
830 IF M=F AND N=G THEN 850
840 F=F-C:G=G-D:GOTO 820
850 NEXT D
860 NEXT C
870 RETURN
880 :
890 GOSUB 470
900 PRINT
910 IF C>H THEN PRINT TAB(9);"I'm the
 champ!"
920 IF H>C THEN PRINT TAB(9);"You're
the champ"
```

```
930 IF H=C THEN PRINT TAB(9);"It's a
draw!"
940 END
950 :
960 REM ** Initialise **
970 CLS
980 X=RND(-TIME)
990 KEY OFF:DEFINT A-Z
1000 X=248:OX=249
1010 DIM A(10,10)
1020 FOR B=2 TO 9
1030 FOR C=2 TO 9
1040 A(B,C)=46
1050 NEXT C
1060 NEXT B
1070 A(5,5)=X:A(6,6)=X
1080 A(6,5)=OX:A(5,6)=OX
1090 P=0
1100 LOCATE 4,8
1110 PRINT "Do you want the first mov
e?"
1120 PRINT TAB(10)"(Y or N)?"
1130 Y$=INKEY$
1140 IF Y$<>"y" AND Y$<>"Y" AND Y$<>"
n" AND Y$<>"N" THEN 1130
1150 CLS
1160 GOSUB 470
1170 IF Y$="y" OR Y$="Y" THEN 340
1180 GOTO 30
```

Testing your speed

The next program, *Reaction Test,* is much shorter than *MSX-thello*, but just as much fun to play. You enter the program, type in RUN, and the message STAND BY appears. After an agonizing wait, STAND BY will vanish, to be replaced with the words "OK, PRESS THE 'Z' KEY!". As fast as you can, you leap for the Z key and press it, knowing that the computer is counting all the time. This is a simpler test than the one in chapter six.

The computer then tells you how quickly you reacted, and compares this with your previous best time. "BEST SO FAR IS..." appears on the screen, and the computer then waits for you to take your hands off the keyboard to prevent cheating (as if you'd do such a thing!) before the whole thing begins again.

```
Stand by


OK, press the 'Z' key!


Your score was 449


Best so far: 152
```

The game continues until you manage to get your reaction time to below 8, which is not an easy task.

Here's the listing of *Reaction Test:*

```
10 REM REACTION TEST
20 HISCRE=1000
30 X=RND(-TIME)
40 IF HISCRE<8 THEN 220
50 CLS
60 PRINT:PRINT:PRINT "Stand by..."
70 FOR A=1 TO 700 + RND(1)*2000
80 NEXT A
90 A$=INKEY$
100 IF A$<>"" THEN 70
110 PRINT:PRINT:PRINT "OK, press the
'Z' key!"
120 COUNT=0
130 COUNT=COUNT+1
140 A$=INKEY$
150 IF A$<>"z" AND A$<>"Z" THEN 130
160 PRINT:PRINT:PRINT "Your score was
"COUNT
170 IF COUNT<HISCRE THEN HISCRE=COUNT
```

```
:BEEP
180 PRINT:PRINT "Best so far is"HISCR
E
190 FOR A=1 TO 1000:NEXT A
200 IF INKEY$<>"" THEN 200
210 GOTO 40
220 PRINT:PRINT "You're the champ!"
230 BEEP
240 END
```

Now you've had a little relaxation, it's time to return to the serious stuff again.

CHAPTER EIGHT

Reading DATA

In this chapter, we'll be looking at three very useful additions to your programming vocabulary: READ, DATA and RESTORE. They are used to get information stored in one part of the program to another part where it can be used.

Enter and run this program, which should make this a little clearer:

```
10 REM READ, DATA and RESTORE
20 DIM A(5)
30 FOR B=1 TO 5
40 READ A(B)
50 PRINT A(B)
60 NEXT B
70 DATA 88,8965,23,-94,3
```

Using line 40, the program READs through the DATA statement in line 70 in order, printing up each item of DATA with line 50.

RESTORE moves the computer back to the *first* item of DATA in the program, as you'll discover if you modify the above program by adding line 55, so it reads as follows:

```
10 REM READ, DATA and RESTORE
20 DIM A(5)
30 FOR B=1 TO 5
40 READ A(B)
50 PRINT A(B)
55 IF B=3 THEN RESTORE
60 NEXT B
70 DATA 88,8965,23,-94,3
```

It does not matter where in the program the DATA is stored. The computer will seek it out, in order from the first item of DATA in the program to the last, as our next program (which scatters the DATA about in an alarming way) convincingly demonstrates:

```
1 DATA 45
10 REM READ, DATA and RESTORE
20 DIM A(5)
22 DATA 888
30 FOR B=1 TO 5
40 READ A(B)
50 PRINT A(B)
55 DATA 432
60 NEXT B
70 DATA 933,254
```

READ and DATA work just as well with string information:

```
10 REM READ/DATA with strings
20 FOR B=1 TO 5
30 READ A$
40 PRINT A$
50 NEXT B
60 DATA test,one,nine,after,noon
```

Note that string DATA does not have to be enclosed within quote marks, unless leading or trailing spaces, and/or punctuation and symbols are significant and must be considered part of the DATA.

You can mix numeric and string DATA within the same program, so long as you take care to ensure that when the program wants a numeric item, a number comes next in the program, and when it wants a string item, it finds it:

```
10 REM READ/DATA
20 FOR B=1 TO 5
30 READ A$:READ A
40 PRINT A$,A
50 NEXT B
60 DATA test,12,one,98767,nine,22,aft
er,-987,noon,.4
```

CHAPTER NINE
Getting Listed

An array is used when you want to create a list of items, and refer to the item by just mentioning the *position* within the list the item occupies. You set up an array by using the command DIM (for dimension). If you type in DIM A(20), the computer will set up a list in its memory called A, and will save space for twenty-one items: A(Ø), A(1), ... and so on ... up to A(2Ø). Each of these items – the A(7) and the rest – are called *elements* of the array.

When you *dimension,* or set up, an array, the computer creates the list in its memory and then fills every item in that list with a zero. So if you told your computer to PRINT A(3) it would print a Ø. You fill the items in an array with a statement like A(2) = 1ØØØ, or by using READ and DATA as we saw in chapter eleven. Once you've given an element a value, you can get the computer to tell you what value the element has by saying PRINT A(n). You can also manipulate the element as though it was the number. That is, A(4)*6 is valid, as is 45 – A(6) and so on.

Your computer will let you use an array of up to 11 elements (that is A(Ø) through to A(1Ø) or TEST(Ø) through to TEST(1Ø)) without having to use the DIM statement first. The moment it comes across a reference to an element of an array, where the *subscript* (the number which follows in parentheses) is between Ø and 1Ø, it automatically creates an array. However, it is good practice to always dimension arrays, even if you are using less than 12 elements.

You may like to 'forget' about the element which has the subscript zero, and pretend that the array starts at one. Many times you'll find it simpler to assume DIM A(8Ø) gives you an array of 8Ø elements (rather than 81 as is the case), and that the first element is A(1).

The first program in this chapter dimensions (sets up, or creates) an array called A with room for sixteen elements. We will ignore the element with the subscript Ø. The B loop, from lines 4Ø to 6Ø, fills the array with random digits between Ø and 9, and then prints them back for you with the loop from 7Ø to 1ØØ (with a slight pause being created by line 9Ø).

Here is the listing:

```
10 REM ARRAYS
20 DIM A(15)
30 CLS:KEY OFF:X=RND(-TIME)
40 FOR B=1 TO 15
50 A(B)=INT(RND(1)*9)
60 NEXT B
70 FOR Z=1 TO 15
80 PRINT "A("Z")  IS "A(Z)
90 FOR T=1 TO 100:NEXT T
100 NEXT Z
```

And here's one example of it in use:

```
A( 1 )  IS  Ø
A( 2 )  IS  3
A( 3 )  IS  5
A( 4 )  IS  3
A( 5 )  IS  3
A( 6 )  IS  2
A( 7 )  IS  6
A( 8 )  IS  4
A( 9 )  IS  2
A( 10 )  IS  4
A( 11 )  IS  2
A( 12 )  IS  Ø
A( 13 )  IS  8
A( 14 )  IS  1
A( 15 )  IS  4
```

This is called a one-dimensional array, because a single digit follows the letter or name which labels the array.

You can also have multi-dimensional arrays, in which more than one number follows the array label after DIM. In our next program, for example, the computer sets up a two-dimensional array called A again, consisting of five elements by five elements (that is, it is dimensioned by DIM A(4,4) as you can see in line 20):

```
100 REM MULTI-DIMENSIONAL ARRAYS
110 DIM A(4,4)
120 CLS:KEY OFF:X=RND(-TIME)
130 FOR B=1 TO 4
140 FOR C=1 TO 4
150 A(B,C)=INT(RND(1)*9)
160 NEXT C
170 NEXT B
180 PRINT "      1  2  3  4"
190 PRINT "                 "
200 FOR B=1 TO 4
210 PRINT B"|";
220 FOR C=1 TO 4
230 PRINT A(B,C);
240 NEXT C
250 PRINT
260 NEXT B
```

When you run it, you'll see something like this:

```
     1  2  3  4
    ------------
1 | 7  7  1  7
2 | 0  8  1  3
3 | 2  2  1  4
4 | 6  5  2  3
```

You specify the element of a two-dimensional array by referring to both its numbers, so the element 1,1 of this array (the element in the top left hand corner of the printout above) is 7 and is referred to as A(1,1). The 1 in the printout is A(3,3), and the 2 below it is A(4,3).

Your computer also supports string arrays. Enter and run the following short program to see string arrays in operation:

```
10 REM STRING ARRAYS
20 DIM A$(5)
30 CLS:KEY OFF:X=RND(-TIME)
40 FOR B=1 TO 5
50 A$(B)=CHR$(INT(RND(1)*26)+65)+CHR$
(INT(RND(1)*26)+65)+CHR$(INT(RND(1)*2
6)+65)+CHR$(INT(RND(1)*26)+65)
60 NEXT B
70 FOR B=1 TO 5
80 PRINT "A$("B")  IS  "A$(B)
90 NEXT B
```

Here's one printout of the program:

```
A$( 1 )  IS  EREZ
A$( 2 )  IS  DZCP
A$( 3 )  IS  PEQY
A$( 4 )  IS  DYHQ
A$( 5 )  IS  FBSJ
```

You can, of course, fill the elements of an array – string or numeric – via DATA or INPUT statements. Here is a string array which is filled by a DATA statement:

```
10 REM STRING ARRAYS
20 DIM A$(5)
30 CLS:KEY OFF:X=RND(-TIME)
40 FOR B=1 TO 5
50 READ A$(B)
60 NEXT B
70 FOR B=1 TO 5
80 PRINT "A$("B") ->   "A$(B)
90 NEXT B
100 DATA THINKING,IS,A,PAINFUL,TASK
```

This is the result of running it:

```
A$( 1 ) ->   THINKING
A$( 2 ) ->   IS
A$( 3 ) ->   A
```

```
A$( 4 ) ->    PAINFUL
A$( 5 ) ->    TASK
```

Escape from Murky Marsh

The next program demonstrates the use of a two-dimensional array for 'holding' an object, and for moving it around within the array. The object in this case is a little shape with a pointed top. The shape is trapped in a murky marsh, and by moving totally at random, it hopes one day to be able to escape from the marsh. The shape is free if it manages to stumble onto the outer rows.

(By the way, the shape in this program demonstrates *Brownian motion,* the random movement shown by such things as tiny particles in a drop of water when viewed under a microscope, or of a single atom of gas in a closed container. Brownian motion explains why a drop of ink gradually mixes into the water into which it has been placed.)

Here is the program listing:

```
10 REM Dragon's Lair
20 X=RND(-TIME)
30 DIM A(10,10)
40 COLOR 1,12
50 CLS:KEY OFF
60 M=0
70 GOSUB 370
80 IF RND(1)>.35 THEN P=P+1 ELSE P=P-
1
90 IF RND(1)>.5 THEN Q=Q+1 ELSE Q=Q-1
100 IF Q<1 THEN Q=1
110 IF Q>10 THEN Q=10
120 IF P<1 THEN P=1
130 IF P>10 THEN P=10
140 M=M+1
150 LOCATE 5,4
160 PRINT "Attempt number"M" "
170 PLAY "ABCDEFG..."
180 A(P,Q)=227
```

```
190 LOCATE 8,7
200 FOR X=1 TO 10
210 FOR Y=1 TO 10
220 PRINT CHR$(A(X,Y));
230 NEXT Y
240 PRINT:PRINT TAB(8);
250 T=INT(RND(1)*9)
260 IF T=0 THEN PLAY "B"
270 IF T=1 THEN PLAY "D"
280 IF T=2 THEN PLAY "E"
290 IF T=3 THEN PLAY "F"
300 IF T=4 THEN PLAY "G"
310 IF T=5 THEN PLAY "A"
320 IF T>5 THEN PLAY "C"
330 NEXT X
340 A(P,Q)=203
350 IF Q<9 OR P<9 THEN 80
360 GOTO 450
370 Q=INT(RND(1)*3)+4
380 P=INT(RND(1)*3)+4
390 FOR X=1 TO 10
400 FOR Y=1 TO 10
410 A(X,Y)=203
420 NEXT Y,X
430 PLAY "O6 L64 T255 V15"
440 RETURN
450 PRINT:PRINT
460 PRINT TAB(4);"Whew...free at last
!!"
470 FOR T=1 TO 2300:NEXT T
480 RUN
```

CHAPTER TEN

The Sound of Music

Your MSX computer contains a very flexible Programmable Sound Generator, which is called the PSG. You trigger the PSG with either the SOUND or the PLAY commands.

The PSG has three independent sound channels (A, B and C) which can be used at once, allowing you to produce three-tone sounds, as you will see (or rather hear) a little later in this chapter. You can also send noise to any channel, rather than pure, musical tones, so you can produce some extremely effective sound effects. You'll be hearing those shortly. Finally, the PSG gives you control of the sound output's *envelope* which allows you to shape the sound's attack, decay and – with some of the envelope options – the 'repeat cycle' of the sound. Our *Envelope Dancer* program demonstrates convincingly the effects which various envelopes can create.

The PSG Registers

The chip at the heart of the PSG is an AY-3-891Ø, and it has 13 registers you can control to produce sounds to your heart's content. Register Ø looks after the fine tuning of sound channel A, and will accept a number between Ø and 255. Register 8 controls the volume of channel A. We turn the volume of channel A up by entering the command SOUND 8,15 in the first program in this chapter, which runs through the pitch possibilities of sound channel A:

```
10 REM Sound channel A
20 SOUND 8,15
30 FOR J=0 TO 255
40 SOUND 0,J
50 FOR T=1 TO 10-J/200:NEXT T
```

```
60 NEXT J
70 SOUND 8,0
```

That program shows the fine pitch tuning of sound channel A. This is, as I said, controlled by register Ø. Register 1 is in charge of coarse pitch tuning for this channel. It can take values up to 15. Here is a program to demonstrate it in action:

```
10 REM Sound channel A (coarse)
20 SOUND 8,15
30 FOR J=1 TO 14
40 SOUND 1,J
50 FOR T=1 TO 100:NEXT T
60 NEXT J
70 SOUND 8,0
```

The fine tuning for channel B is controlled by register 2, and register 9 controls channel B's volume. If both channel A and B are turned on, and a note is played through each of the channels, a two-tone chord (slightly out of tune, in this case) can be produced:

```
10 REM  A and B together
20 SOUND 8,15:SOUND 9,15
30 SOUND 1,2
40 SOUND 3,6
50 FOR J=1 TO 900:NEXT J
60 SOUND 8,0:SOUND 9,0
```

Line 5Ø, in the program above, allows the tone to sound for a while, before turning it off in line 6Ø by putting zero into each of the volume registers.

If we modify this program a little, we can produce a series of rather unlovely, two-note chords from channels A and B:

```
10 REM  A and B together
20 SOUND 8,15:SOUND 9,15
30 FOR J=1 TO 14
40 SOUND 1,J
50 SOUND 3,14-J
60 FOR T=1 TO 200:NEXT T
```

```
70 NEXT J
80 SOUND 8,0:SOUND 9,0
```

We can now turn to the third channel, C. The volume for C is controlled by register 10, while fine pitch tuning is controlled by register 4 and coarse pitch tuning by register 5. The three sound channels, if triggered together, allow us to produce three-note chords (which true musicians may well wish they had never heard):

```
10 REM  A, B & C together
20 SOUND 8,15:SOUND 9,15:SOUND 10,15
30 FOR J=1 TO 14
40 SOUND 1,J
50 SOUND 3,14-J
60 SOUND 5,2*J/3
70 FOR T=1 TO 200:NEXT T
80 NEXT J
90 SOUND 8,0:SOUND 9,0:SOUND 10,0
```

The above program uses the coarse pitch tuning of the three channels. Our next program generates chords at random, using the fine pitch tuning. It prints the values on the screen so you can tell which numbers are producing the result you hear. Press any key to stop the program:

```
10 REM Random chord creation
20 CLS:KEY OFF
30 PRINT:PRINT TAB(6);"Press any key
to stop"
40 :
50 SOUND 8,15:SOUND 9,15:SOUND 10,15
60 X=RND(-TIME)
70 :
80 X=INT(RND(1)*255)
90 Y=INT(RND(1)*255)
100 Z=INT(RND(1)*255)
110 LOCATE 12,5
120 PRINT "X ="X"   "
130 PRINT:PRINT TAB(12);"Y ="Y"   "
140 PRINT:PRINT TAB(12);"Z ="Z"   "
150 IF RND(1)>.5 THEN SOUND 0,X
```

```
160 IF RND(1)>.5 THEN SOUND 2,Y
170 IF RND(1)>.5 THEN SOUND 4,Z
180 FOR T=1 TO RND(1)*50+30:NEXT T
190 IF RND(1)>.5 THEN 150
200 IF INKEY$<>"" THEN 230
210 GOTO 80
220 :
230 SOUND 8,0:SOUND 9,0:SOUND 10,0
```

Enter and run the following program, which produces the by-now familiar sound of random 'music' from channel A (ignore line 30 for the moment, as we haven't discussed register 6 yet):

```
10 REM Noise
20 SOUND 8,15
30 SOUND 6,INT(RND(1)*64)
40 SOUND 0,INT(RND(1)*256)
50 FOR T=1 TO 300:NEXT T
60 IF INKEY$<>"" THEN 80
70 GOTO 30
80 SOUND 8,0
```

Now modify that program so it reads as follows, adding line 25 and modifying line 70:

```
10 REM Noise
20 SOUND 8,15
25 SOUND 7,INT(RND(1)*15)
30 SOUND 6,INT(RND(1)*64)
40 SOUND 0,INT(RND(1)*256)
50 FOR T=1 TO 300:NEXT T
60 IF INKEY$<>"" THEN 80
70 GOTO 25
80 SOUND 8,0
```

This variation of the *Noise* program uses register 6, which modulates the noise channel, and will accept values from zero to 63.

You will only become expert at using the PSG by playing around with it, and taking note of the ways various effects are produced. I

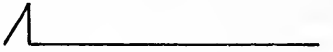
suggest you keep a sheet of paper to note particularly effective sounds when you find them, so you can incorporate them into your programs when you have a need for a good effect.

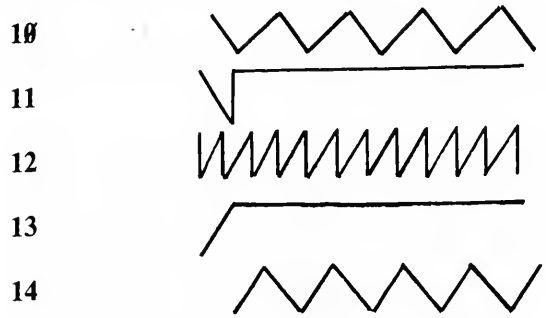
Although we'll be discussing envelopes in a moment, and the ways in which these envelopes can 'bend' the sound, it is worth noting that quite extraordinary effects can be generated with the simplest means, using the PSG. For example, our next program, *Electronic Music,* uses nothing but a loop between two randomly chosen values, sticking numbers in the fine pitch tuning register (Ø) for channel A. However, the results are staggering, as your ears will testify:

```
100 REM ELECTRONIC MUSIC
110 :
120 SOUND 8,6
130 A=RND(1)*255
140 B=RND(1)*255
150  IF A>B OR RND(1)>.96 THEN 140
160 :
170 FOR J=B TO A STEP -1
180  SOUND Ø,J
190 NEXT J
200 FOR Z=1 TO 200:NEXT Z
210 GOTO 130
```

Envelopes

There are eight different envelopes available on your MSX computer. The number you put into register 13 can be from zero to 15, and the effects the various numbers produce is indicated by the diagrams below:

Value:	**Envelope produced:**
Ø,1,2,3,9	
4,5,6,7,15	
8	



The next program demonstrates the PSG envelopes in action. You'll see that envelopes 8, 10, 11, 12, 13 and 14 are ones which continue. They produce steady notes, from a quiet start with 13, but the loudness of the sound rises and falls (at differing rates) with some of the others:

```
10 REM Envelopes
20 CLS:KEY OFF
30 SEED=RND(-TIME)
40 FOR X=0 TO 14
50 SOUND 13,X
60 PRINT:PRINT "This is envelope"X
70 SOUND 0,INT(RND(1)*156)
80 SOUND 8,16:SOUND 12,10
90 IF INKEY$="" THEN 90
100 NEXT X
110 GOTO 40
```

Our next program chooses an envelope at random, and puts its shape on the screen (so you do not have to refer to our table to see the shape of the various envelopes). Press any key to get a new envelope:

```
10 REM Envelope Dancer
20 CLS:KEY OFF
30 SEED=RND(-TIME)
40 E=INT(RND(1)*15)
50 PITCH=INT(RND(1)*16)
60 PRINT:PRINT:PRINT
70 PRINT TAB(5);"Envelope:"E
80 PRINT TAB(9);"Pitch:"PITCH
```

```
90 GOSUB 160
100 SOUND 13,E
110 SOUND 1,PITCH
120 SOUND 8,16:SOUND 12,16
130 IF INKEY$="" THEN 130
140 GOTO 40
150 :
160 REM Envelope shape
170 IF E<>0 AND E<>1 AND E<>2 AND E<>
3 AND E<>9 THEN 200
180 PRINT TAB(7);"\_____"
190 RETURN
200 IF E<>4 AND E<>5 AND E<>6 AND E<>
7 AND E<>15 THEN 230
210 PRINT TAB(7);"/|____"
220 RETURN
230 IF E<>8 THEN 260
240 PRINT TAB(7);"\/\/\/\"
250 RETURN
260 IF E<>10 THEN 290
270 PRINT TAB(7);"\ / \ / \ /"
280 RETURN
290 IF E<>11 THEN 330
300 PRINT TAB(7);"  _______"
310 PRINT TAB(7);"\|"
320 RETURN
330 IF E<>12 THEN 360
340 PRINT TAB(7);"/\/\/\/\/"
350 RETURN
360 IF E<>13 THEN 400
370 PRINT TAB(8);"________"
380 PRINT TAB(7);"/"
390 RETURN
400 PRINT TAB(7);"/ \ / \ / \"
410 RETURN
```

Acting Macro

Now while it's all very well producing music, and strange sound effects with the aid of numbers, it is much simpler – at least for

musicians – to refer to notes by their names. The MSX computer's *Music Macro Language* allows you to do this.

Enter the following program, which repeats the scale of C major, fairly slowly, over and over again:

```
10 REM SCALE
20 PLAY "O4 CDEFGAB O5 C.."
100 GOTO 20
```

As is pretty obvious from that short program, the command PLAY is used (rather than SOUND) when using the *Macro Language* commands. The note names or letters (A for the note A and so on) are also pretty self-explanatory. However, what is the "Ø4" and "Ø5" we see in line 2Ø? And those dots after the final C. What are they there for? They are just a few of the possible commands you can use when programming in the music language.

The note names A to G produce the relevant note, while a hash symbol (#) or a + will raise the note a semitone (to get, say, F sharp, you enter F# or F+) and a minus sign(–) will produce a flat (so B flat is B–). These only work if there is a black note on a piano in that position (so F flat does not work). The letter Ø, such as we saw in the brief *C major scale* program, controls the *octave* in which the sound will be heard. There are eight octaves (numbers 1 to 8) and the computer defaults to octave 4 if one is not specified.

Note, by the way, that the spaces between the "Ø4" and the note names are there just to make it easier to see what is in the relevant string following PLAY. The spaces are ignored by your MSX computer when actually playing the music.

Instead of note names, you can use the letter N followed by a number (from Ø to 96). Zero tells your MSX machine that you want a rest, and 1 equals the lowest C the computer can play, right up to 96 which is the highest note you can get from your computer.

As well as controlling the pitch of the note played, the *Macro* language allows you to control the speed at which your music is made, and the length of the individual notes which form it. The letter "L" is followed by a number (as "L64") to determine the length. L1

is a whole note, L4 a quarter note, right up to L64 which is a sixty-fourth note. If you do not specify an L command, and one has not been specified previously in the current run, the computer defaults to 4.

Enter and run the following program, which shows L in use:

```
10 REM SCALE SHOWING L CHANGE
15 SEED=RND(-TIME)
16 CLS:PRINT "LENGTH IS"4
20 PLAY "O4 CDEFGAB O5 C.."
25 IF INKEY$="" THEN 25
30 X=INT(RND(1)*5)+1
40 ON X GOSUB 110,120,130,140,150
50 PRINT:PRINT "LENGTH IS"L
100 GOTO 20
110 PLAY "L1":L=1
115 RETURN
120 PLAY "L2":L=2
125 RETURN
130 PLAY "L4":L=4
135 RETURN
140 PLAY "L8":L=8
145 RETURN
150 PLAY "L64":L=64
155 RETURN
```

The speed, or tempo, of the music, is determined by the number which follows the letter T within a music string. It controls the number of quarter notes which the MSX machine will play in one minute. It defaults to 120, and can be modified by you from 32 to 255. In the next program, a variation of the previous one, we look at the use of T to modify the tempo. Before you run the program, type in – as a direct command – PLAY "L4" and then press the ENTER key:

```
10 REM SCALE WITH L AND T
15 SEED=RND(-TIME)
16 CLS:PRINT "LENGTH IS"4:PRINT "TEMP
O IS 120"
```

```
20 PLAY "O4 CDEFGAB O5 C.."
25 IF INKEY$="" THEN 25
30 X=INT(RND(1)*5)+1
40 ON X GOSUB 110,120,130,140,150
50 PRINT:PRINT "TEMPO IS"T
100 GOTO 20
110 PLAY "T32":T=32
115 RETURN
120 PLAY "T64":T=64
125 RETURN
130 PLAY "T120":T=120
135 RETURN
140 PLAY "T156":T=156
145 RETURN
150 PLAY "T255":T=255
155 RETURN
```

You can modify other aspects of the string to control your music. V looks after the volume, and runs from Ø to 15, with a default value of 8. R is used to get a rest, and is followed by the same numbers as control the L, length, command. You can bring the envelope into PLAY commands, with the letter S, which must be followed by a number from Ø to 15. It affects all three channels at once, and defaults to 1 (which starts a note instantly at full intensity, and quickly fades away).

The letter M triggers the command which controls the *period* of the envelope. The period is the time it takes the sound to go from zero to full intensity, and this defaults to 255.

As I guess you can imagine, the string following the command PLAY can get pretty confusing and crowded. One way to keep it reasonably clear what is happening is to use spaces within the strings, as you have seen me doing in this chapter. As I pointed out earlier, the computer ignores spaces, but they can be a great aid in helping you decipher a string when you come back to it after a break.

PLAY strings can be further simplified by substituting portions of them with assigned variables. You need the letter X within a string to tell the computer that the following information is an assigned variable, rather than letter to be interpreted directly (i.e. that "A" is

a numeric variable with a value, and not just the note A). The end of an assigned variable is signalled by a semi-colon (;) within the string. The following two programs should help make this clear.

First enter and run this program, which plays a fairly crummy version of *Twinkle, Twinkle Little Star:*

```
10 REM Twinkle twinkle
20 PLAY "O3 L8 T200 V15"
30 PLAY "CR8C. O4 GR8G. AR8A. G. R4"
40 PLAY "FR8F. ER8E. DR8D. C.... R8"
45 FOR H=1 TO 2
50 PLAY "GR8G. FR8F. ER8E. D.... R8"
60 NEXT H
70 PLAY "CR8C. O4 GR8G. AR8A. G. R4"
80 PLAY "FR8F. ER8E. DR8D. C.... R4"
```

If you look carefully at that listing, you'll see there are some segments which are repeated while the tune is played. Line 3Ø is repeated exactly in line 7Ø, and line 4Ø is repeated, apart from the final rest, in line 8Ø. (Note also that line 5Ø is held within a FOR/NEXT loop, to play it twice.) We will assign the strings A$, B$ and C$ to the repeated sections of this song, and then use them within the program as follows:

```
10 REM Twinkle twinkle II
20 PLAY "O3 L8 T200 V15"
30 A$="CR8C. O4 GR8G. AR8A. G. R4"
40 B$="FR8F. ER8E. DR8D. C.... R8"
50 C$="GR8G. FR8F. ER8E. D.... R8"
60 PLAY "XA$;XB$;XC$;"
70 PLAY "XC$;XB$;XA$;XB$;"
80 PLAY "XA$;XB$;XC$;"
90 PLAY "XB$;"
```

You can see in lines 6Ø to 9Ø that the assigned strings are played, and are identified as such to the computer by the preceding X and the following semi-colon. Make sure that your assigned string is not longer than 255 characters, or you'll discover the final notes are not played.

Finally, we need to explain the dot (.) which has appeared in several programs, such as in the lines 3Ø, 4Ø and 5Ø of *Twinkle II.* A dot after a note causes it to be played one and a half times as long as it would have been otherwise; two dots means the note (or rest) lasts for two and a quarter times as long as it would have otherwise; and three dots increases the length of the note to almost four times (actually 3.375) its original length. You can keep adding dots to increase the length of a note or a rest.

Finally in this chapter, let's bring together a lot of what we have discovered by turning our computer into a sort of automatic piano, and getting it to write a continuously changing piece of music. *The Jazz Player* program, in fact, does not produce very tuneful music, but it is a great demonstration program, and comes complete with a screen display which shows musical notes moving up the screen, with random colour changes. When your inspiration is fading, you can run this program to see electronic creativity in action:

```
10 REM THE Jazz Player
20 SEED=RND(-TIME)
30 CLS:KEY OFF
40 DIM A$(16)
50 PLAY "T120 L8 S11M1999"
60 FOR G=1 TO 10
70 READ A$(G)
80 NEXT G
90 W=INT(RND(1)*12)+3
100 P=INT(RND(1)*12)+3
110 Q$=A$(W+INT(RND(1)*3))
120 R$=A$(W)
130 T$=A$(W-INT(RND(1)*3))
140 REP=RND(1)
150 FOR Y=1 TO RND(1)*4
160 PLAY "XQ$;","XR$;","XT$;"
170 PLAY A$(P),A$(P+1),A$(P-1)
180 IF REP>.4 THEN PLAY "XT$;","XQ$;"
190 NEXT Y
200 IF W>8 THEN 90
210 COLOR INT(RND(1)*13+2),INT(RND(1)
*13+2)
220 K=INT(RND(1)*25)+5
```

```
230 PRINT TAB(K);CHR$(222)
240 PRINT TAB(K);CHR$(222)
250 PRINT TAB(K);CHR$(219)
260 GOTO 90
270 DATA "C","D","03","F","G","04","B
","C.","E","05","A","A#...","E.","G."
,"F#","C....."
```

CHAPTER ELEVEN

Functional Fun

Despite the title of this chapter, functions are not really much fun, but as they are vital elements in your MSX programming quiver, it's important to know about them. Once you've set up a function, using the DEF FN statement, you can trigger the function at will within a program. Functions actually allow you to define new words which become part of your programming language for that program.

A function acts on a number (or variable) to return a value. When the name you've assigned to a function is included within a program, the relevant number of variable must follow the name.

The function is made up of a name and a definition. This should be clear if you enter, and run, the following program:

```
10 REM DEFINE FUNCTION DEMO
20 CLS:KEY OFF
30 DEF FNHALF(X)=X/2
40 INPUT "Enter X";X
50 Z=FNHALF(X)
60 PRINT TAB(5);Z
70 PRINT
80 GOTO 40
```

The function is defined in line 30. First you'll see DEF FN which tells the computer that this line is to defining the function which follows. The *name* of the function here is HALF and the *definition,* which follows the equals sign, is X/2. You'll see that after the name HALF there is an X in brackets. This is a dummy variable which tells the computer how a variable within a program – which will be processed by the definition – will be presented.

Here's the above program in action:

```
Enter X? 43
       21.5

Enter X? 45.984
       22.992

Enter X? 123455232
       61727616
```

The roles of the various parts of a function, the dummy variable, the name and the definition, may be more clear to you after you enter and run this program, which generates random numbers between 1 and the value entered:

```
10 REM DEFINE FUNCTION DEMO
20 CLS:KEY OFF
30 DEF FNRAN(X)=INT(RND(1)*X)+1
40 INPUT "Enter X";X
50 Z=FNRAN(X)
60 PRINT TAB(5);Z
70 PRINT
80 GOTO 40
```

This is an example of it in use:

```
Enter X 45
       27

Enter X 1234
       132

Enter X 99999
       76597

Enter X 23
       14

Enter X 12
       9
```

You may well find that a function like this is a boon in any program in which many, many random numbers needed to be generated.

The function can be used directly in a program, as you'll see in the next program:

```
10 REM DEF FN in program
20 CLS:KEY OFF
30 DEF FNRAN(X)=INT(RND(1)*X)+1
40 INPUT "Enter X";X
50 FOR J=1 TO FNRAN(X)
60 PRINT J;
70 NEXT J
80 PRINT:PRINT
90 GOTO 40
```

Here it is in action:

```
Enter X 12
 1  2  3  4  5  6  7  8

Enter X 12
 1  2

Enter X 12
 1  2  3  4  5  6  7  8  9  10

Enter X 21
 1  2  3  4  5  6  7  8  9  10  11  1
2  13
```

Finally in this chapter, here's a summary of the major mathematical symbols and functions used in MSX BASIC:

Usual symbol:	Computer symbol:
+ (plus)	+
− (minus)	−
× (multiply)	*
÷ (divide)	/
m^n	m^n

The mathematical functions include:

Computer word:	Meaning:
ATN	ARCTANGENT
SIN	SINE
COS	COSINE
TAN	TANGENT
INT	Reduce to next lowest whole number
SGN	Sign (returns –1 if negative, Ø if zero, 1 if positive)
ABS	Returns number without its sign (so ABS (–5) is 5)
SQR	Square root
LN	Natural log

CHAPTER TWELVE

Adding Life to Programs

The MSX computers have many facilities, and you should make the most of them. In this chapter of the book, we are going to take a simple program, and then elaborate it by adding such things as sound. Once you've worked through this chapter, you should find it easy to apply the ideas to other programs you are writing.

The program we're going to use as the core of our development work is a 'Duck Shoot' one, in which little objects fly across the screen, and you have to try and shoot them down.

In the first version of the program, the little objects are letters chosen at random, and you are the letter "X". You fire at the 'ducks' by pressing the "F" key. You move yourself left using the "Z" key and to the right with the "M" key. CAPS LOCK must be engaged before you run the program.

Although there is no time limit within the program, so you do not have to shoot all the ducks as quickly as you can, there is a limit on the number of shots you can fire. In some of the versions of this game in this part of the book, you'll see (line 5Ø) that the program starts with a limit of 15 shots. In the other more complex versions, you have many more shots. The number of shots is deliberately kept low in the first versions so you will not be able to get a high score just by leaving your finger on the "F" key and waiting for the ducks to fly into the line of fire.

Here, then, is the first program. Type it into the computer and then run it:

```
10 REM Duck Shoot
20 CLS:KEY OFF
30 DEFINT A-Z
40 SCRE=0
50 SHOTS=15
60 A$="  SSSG ZAB     DK  SL DF G     F
D FGGG"
70 ACROSS=19
80 DOWN=19
90 REM ** Main game **
100 LOCATE 1,11:PRINT A$
110 LOCATE ACROSS-1,DOWN:PRINT " X "
120 B$=INKEY$
130 IF B$="F" THEN SHOTS=SHOTS-1:IF M
ID$(A$,ACROSS,1)<>" " THEN SCRE=SCRE+
57:MID$(A$,ACROSS,1)=" "
140 LOCATE 1,5:PRINT "Score:"SCRE,"
  Shots left:"SHOTS" "
150 IF SHOTS<1 THEN LOCATE 1,11:PRINT
 "---- That's the end of the game ---
-":GOTO 200
160 IF B$="Z" THEN ACROSS=ACROSS-1
170 IF B$="M" THEN ACROSS=ACROSS+1
180 A$=MID$(A$,2)+LEFT$(A$,1)
190 GOTO 100
200 GOTO 200
```

You'll see the letters which are held in A$ (see line 60) moving across near the top of the screen. You (the "X") will be further down the screen, about a quarter of the way across. You can – as I mentioned a few moments ago – move yourself back and forth using the "Z" and "M" keys to get yourself into the position which you think gives you the best possible chance.

When you judge a 'duck' is directly overhead, press the "F" key to fire your patented anti-duck missile. The number after the words "Shots left" (near the top right hand corner of the screen) will decrease and, if you have been accurate, the number after the word "Score" will increase.

Note, by the way, that I have deliberately used *explicit names* for the variables within this program. That is, the variable name for the

score is SCRE, for the shots it is SHOTS and for your position across the screen, the variable name is ACROSS. Even though it takes a little longer to type long variable names into a program, the advantages of using explicit names to keep the purpose of various parts of the listing clear outweighs the extra time it takes to type them in.

If, for example, you were writing a program like this, and you decided it would be better if the "X" was printed further down the screen, you would not have to search through the program to work out which variable held your 'down' co-ordinate. If you had used explicit names, as in this case, you would find it very easy to locate the variable you were looking for.

Run the DUCK SHOOT program a few times, then return to the book for the first part of a discussion on it.

Line 6Ø defines the string variable A$ as a long series of letters and spaces. The letters can be anything you like; do not feel you have to copy mine. The important thing, however, is that the string is 32 characters long. You can check this by running program briefly, stopping it, then typing in PRINT LEN (A$). If your string is the correct length, PRINT LEN (A$), followed by RETURN, will give you the answer 32.

The appearance of movement given to the ducks is created by the string-handling commands which were explained a little earlier in the book. Refer back to this section now if you need to remind yourself how they work.

The vital line for movement is 18Ø, which resets A$ equal to all of the string without its first character – that is to MID$(A$,2) – and then adds to the very end of it the character of the string which was at the beginning, LEFT$(A$,1). The string is reprinted over and over again at 1, 11 (see line 1ØØ, where LOCATE 1, 11 is used to move the cursor to the position where we next wish to print) which is eleven lines down, and at the first position across the line.

Because the string is, in effect, being 'shifted along' one character at a time before it is reprinted, the elements within the string appear to move smoothly along. Using strings in this way is one of the simplest ways on the computer to create smoothly moving graphics.

The string handling also makes it very simple to cause the shot duck to disappear from the sky. As the string is 32 characters long, each character 'shot' can be referred to as MID$(A$,ACROSS,1).

Look at line 13Ø. When the computer comes across an IF/THEN statement – as you know – it checks to see if it is true. If it finds that it is not true, then it moves along to the next line in the program, without bothering to carry out any further instructions which may be on the same line. Note that before the computer gets to line 13Ø, it sets B$ equal to the key you are pressing (INKEY$). If the computer finds, at the start of line 13Ø, that B$ does not equal "F" (as will be the case when you are not pressing the "F" key), then it proceeds to line 14Ø, missing all the information and instructions which follow the IF B$ = "F" ... line.

If, however, you are pressing "F" when the computer reads the keyboard, the computer continues working through the line, and reduces the value of the variable SHOTS by one. Then it hits another IF/THEN condition in which it looks at the element of A$ which is directly above the position of the "X" at that moment.

If line 13Ø discovers that this element of A$ is anything but a space, you have hit a duck, so the computer continues working through the line. The variable SCRE is incremented by 57 and, finally in line 13Ø, that particular element of A$ is set to a blank, so the duck disappears.

Now, this takes some time to explain, but you'll find the computer does it apparently instantaneously. You press "F", the score increases by 57 (if you're a good shot), the number of shots left drops by one, and the duck disappears. You'll see (line 15Ø) that the game continues until you run out of shots, when the game terminates. Take note of your score at this point, and see if you can beat it on subsequent runs.

Once you have the program running to your satisfaction, and you have a pretty good idea of how it works, modify to read like the following program. You do not have to NEW the computer. Just add line 15, and the BEEP at the end of line 13Ø:

```
10 REM Duck Shoot - II
15 COLOR 12,15:REM Green on white
20 CLS:KEY OFF
30 DEFINT A-Z
40 SCRE=0
50 SHOTS=15
60 A$="  SSSG ZAB     DK   SL DF G     F
D FGGG"
70 ACROSS=19
80 DOWN=19
90 REM ** Main game **
100 LOCATE 1,11:PRINT A$
110 LOCATE ACROSS-1,DOWN:PRINT " X "
120 B$=INKEY$
130 IF B$="F" THEN SHOTS=SHOTS-1:IF M
ID$(A$,ACROSS,1)<>" " THEN SCRE=SCRE+
57:MID$(A$,ACROSS,1)=" ":BEEP
140 LOCATE 1,5:PRINT "Score:"SCRE,"
  Shots left:"SHOTS" "
150 IF SHOTS<1 THEN LOCATE 1,11:PRINT
 "---- That's the end of the game ---
-":GOTO 200
160 IF B$="Z" THEN ACROSS=ACROSS-1
170 IF B$="M" THEN ACROSS=ACROSS+1
180 A$=MID$(A$,2)+LEFT$(A$,1)
190 GOTO 100
200 GOTO 200
```

As you'll see, even these tiny changes improve the program considerably.

Instead of just the BEEP at the end of line 13Ø, we can now use the sound more effectively. Modify your program so it reads as follows and run it. You'll find the game begins with a slow upward sweep of 'music' which gets faster as the note rises. Line 135 ensures that there'll be a gradually rising tone throughout the game. The other note you hear is related to the actual letter which forms the leftmost 'duck':

```
10 REM Duck Shoot
15 COLOR 12,15:REM Green on white
20 CLS:KEY OFF
```

```
30 DEFINT A-Z
34 SOUND 0,1:SOUND 1,1:SOUND 8,15
35 FOR G=255 TO 0 STEP -(RND(1)*5+1)
36 SOUND 0,G:FOR T=1 TO G/3:NEXT T:NE
XT G
40 SCRE=0
50 SHOTS=15
60 A$="  SSSG ZAB     DK   SL DF G     F
D FGGG"
70 ACROSS=19
80 DOWN=19
90 REM ** Main game **
100 LOCATE 1,11:PRINT A$
101 NTE=ASC(A$):SOUND 8,(RND(1)*5)+10
102 IF NTE<>32 THEN SOUND 0,2.7*NTE
110 LOCATE ACROSS-1,DOWN:PRINT " X "
120 B$=INKEY$
130 IF B$="F" THEN SHOTS=SHOTS-1:IF M
ID$(A$,ACROSS,1)<>" " THEN SCRE=SCRE+
57:MID$(A$,ACROSS,1)=" ":SOUND 0,SHOT
S+3
135 SOUND 8,10:SOUND 0,SHOTS+3
140 LOCATE 1,5:PRINT "Score:"SCRE,"
  Shots left:"SHOTS" "
150 IF SHOTS<1 THEN LOCATE 1,11:PRINT
 "---- That's the end of the game ---
-":GOTO 200
160 IF B$="Z" THEN ACROSS=ACROSS-1
170 IF B$="M" THEN ACROSS=ACROSS+1
180 A$=MID$(A$,2)+LEFT$(A$,1)
190 GOTO 100
200 SOUND 8,0
210 GOTO 210
```

We can continue to work on the program. In our next version, the line of letters has been replaced by a line of randomly-generated characters, like these:

As well, the program will stop if you manage to destroy all the ducks

(see line 15Ø) whether or not you have used up all your shots. Whenever you hit a duck, it will turn into a black blob before it vanishes. As you can see, we are creating quite a worthwhile program from a very simple beginning.

```
 10 REM Duck Shoot
 12 X=RND(-TIME)
 15 COLOR 12,15:REM Green on white
 20 CLS:KEY OFF
 30 DEFINT A-Z
 34 SOUND 0,1:SOUND 1,1:SOUND 8,15
 35 FOR G=255 TO 0 STEP -(RND(1)*5+1)
 36 SOUND 0,G:FOR T=1 TO G/3:NEXT T:NE
XT G
 40 SCRE=0
 50 SHOTS=50
 60 GOSUB 1000
 70 ACROSS=19
 80 DOWN=19
 90 REM ** Main game **
100 LOCATE 1,11:PRINT A$
101 NTE=ASC(A$):SOUND 8,(RND(1)*5)+10
102 IF NTE<>32 THEN SOUND 0,.7*NTE
110 LOCATE ACROSS-1,DOWN:PRINT " ";CH
R$(216);" ":REM Little triangle
120 B$=INKEY$
130 IF B$="F" THEN SHOTS=SHOTS-1:IF M
ID$(A$,ACROSS,1)<>" " THEN SCRE=SCRE+
57:MID$(A$,ACROSS,1)=CHR$(219):SOUND
0,SHOTS+3:LOCATE 1,11:PRINT A$:MID$(A
$,ACROSS,1)=" "
135 SOUND 8,10:SOUND 0,SHOTS+3
140 LOCATE 1,5:PRINT "Score:"SCRE,"
  Shots left:"SHOTS" "
149 REM 32 SPACES IN NEXT LINE FOR A$
150 IF SHOTS<1 OR A$="
                    " THEN LOCATE 1,11:P
RINT "---- That's the end of the game
 ----":PRINT:PRINT ,"Rating:"197*SHOT
S:GOTO 200
160 IF B$="Z" THEN ACROSS=ACROSS-1
```

```
170 IF B$="M" THEN ACROSS=ACROSS+1
180 A$=MID$(A$,2)+LEFT$(A$,1)
190 GOTO 100
200 SOUND 8,0
210 GOTO 210
1000 REM DEFINE A$
1010 A$="":DIM C(10)
1020 FOR B=1 TO 10
1030 READ C(B)
1040 NEXT B
1050 FOR T=1 TO 32
1060 IF RND(1)>.7 THEN A$=A$+CHR$(C((
RND(1)*10+1))) ELSE A$=A$+" "
1070 NEXT T
1080 RETURN
1100 DATA 191,153,215,234,243,236,188
,199,157,210
```

As you can see, the program listing is getting quite complex as the program develops. You may well wish to follow a similar series of steps when creating your own programs. Get a raw 'basic' program up and running, and then add to it and modify it, until you have created a masterpiece. On some computers, adding all these extras could slow things down considerably, but your MSX computer works so quickly the extra tasks you have given it appear to make very little difference to the running speed.

We come now to our final version of the game. You should be able to work out how each part works, from the information gained from the other sections. Note that you now have four lines of ducks, two going from left to right, and two from right to left. You must get rid of all of them in order to get a 'rating'. Even if you found it relatively easy to demolish all the ducks in the earlier versions before you ran out of ammunition, you'll find it is not so with this version:

```
10 REM Duck Shoot
12 X=RND(-TIME)
15 COLOR 12,15:REM Green on white
20 CLS:KEY OFF
30 DEFINT A-Z
34 SOUND 0,1:SOUND 1,1:SOUND 8,15
```

```
35 FOR G=255 TO 0 STEP -(RND(1)*5+5)
36 SOUND 0,G:FOR T=1 TO G/3:NEXT T:NE
XT G
40 SCRE=0
50 SHOTS=100
60 GOSUB 1000
70 ACROSS=19
80 DOWN=19
90 REM ** Main game **
93 LOCATE 1,12:PRINT Z$
94 LOCATE 1,8:PRINT Z$
95 LOCATE 1,9:PRINT A$
100 LOCATE 1,11:PRINT A$
101 NTE=ASC(A$):SOUND 8,(RND(1)*5)+10
102 IF NTE<>32 THEN SOUND 0,.7*NTE
110 LOCATE ACROSS-1,DOWN:PRINT " ";CH
R$(216);" ":REM Little triangle
120 B$=INKEY$
130 IF B$="F" THEN SHOTS=SHOTS-1:IF M
ID$(A$,ACROSS,1)<>" " THEN SCRE=SCRE+
57:MID$(A$,ACROSS,1)=CHR$(219):SOUND
0,SHOTS+3:LOCATE 1,11:PRINT A$:MID$(A
$,ACROSS,1)=" "
132 IF B$="F" THEN SHOTS=SHOTS-1:IF M
ID$(Z$,ACROSS,1)<>" " THEN SCRE=SCRE+
57:MID$(Z$,ACROSS,1)="#":SOUND 0,SHOT
S+3:LOCATE 1,9:PRINT Z$:MID$(Z$,ACROS
S,1)=" "
135 SOUND 8,10:SOUND 0,SHOTS+3
140 LOCATE 1,5:PRINT "Score:"SCRE,"
  Shots left:"SHOTS" "
149 REM 32 SPACES IN NEXT LINE FOR A$
150 IF SHOTS<1 OR A$="
                    " AND Z$="
                          " THEN LOCATE 1
,11:PRINT "---- That's the end of the
 game ----":PRINT:PRINT ,"Rating:"197
*SHOTS:GOTO 200
160 IF B$="Z" THEN ACROSS=ACROSS-1
170 IF B$="M" THEN ACROSS=ACROSS+1
180 A$=MID$(A$,2)+LEFT$(A$,1)
```

```
185 Z$=MID$(Z$,32)+LEFT$(Z$,31)
190 GOTO 93
200 SOUND 8,0
201 LOCATE 1,8:PRINT "
                      "
202 LOCATE 1,9:PRINT "
                      "
203 LOCATE 1,12:PRINT "
                       "
210 GOTO 210
1000 REM DEFINE A$,Z$
1010 A$="":Z$="":DIM C(10)
1020 FOR B=1 TO 10
1030 READ C(B)
1040 NEXT B
1050 FOR T=1 TO 32
1060 IF RND(1)>.7 THEN A$=A$+CHR$(C((
RND(1)*10+1))) ELSE A$=A$+" "
1065 IF RND(1)>.7 THEN Z$=Z$+CHR$(C((
RND(1)*10+1))) ELSE Z$=Z$+" "
1066 SOUND 0,3*T
1070 NEXT T
1080 RETURN
1100 DATA 191,153,215,234,243,236,188
,199,157,210
```

CHAPTER THIRTEEN
Graphics Galore

One of the most outstanding aspects of MSX BASIC is its wide range of graphics commands. You can do things easily on your MSX machine which can only be achieved by extremely skilled (and painstaking) programmers on many other computers. In this chapter, we'll be going through the important graphic commands, producing a range of dramatic demonstrations and displays, and giving you the tools to create dazzling visuals of your own. It is vital that, as in other chapters within this book, you enter the sample programs as you come to them.

First, you need to know that there are three screen *types* on your MSX computer. One of these is the *border,* the next is called the *character pattern screen* and the third type (of which you can have up to 32) is the *sprite screen.* The computer knows which screen type to use, and assigns the information you give it automatically to the correct type.

On the character pattern screen, the computer prints in both text and graphics modes. When you list a program (or when you first turn on the computer), you're looking at the character pattern screen. It is almost certain that what you see on your screen right now is material printed on the character pattern screen.

Once we've looked at the marvellous possibilities of the character pattern screen in this chapter, we'll move onto sprites in the next chapter. The sprite screens (and, remember, there can be 32 of them if you wish) are designed to make moving graphics – like those you see in arcade games – easy to create and manipulate.

The border, when it is visible, appears at the top and bottom of the

picture, and cannot be used for printing text or graphics. It acts as a frame to the screen which you are using at any time.

To look after all these screens, the computer has a *priority* system. The higher priority screens cover up the lower ones. The border is the lowest priority screen, followed by the character pattern screen, and the sprites screens have the highest priority. You give each sprite screen an identifying number (from Ø to 31) and the computer assigns them priorities in terms of this number, with Ø having the highest sprite screen priority, and 31 having the lowest.

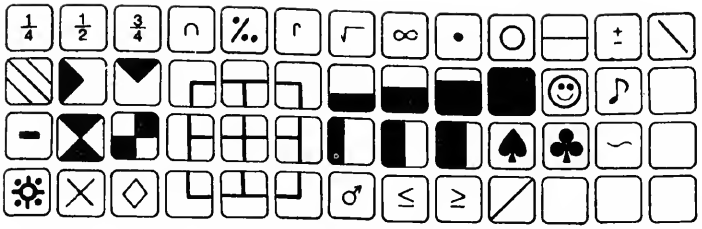
The Silver SCREEN

The SCREEN command is used to specify which *version* of the character pattern screen you wish to use. The text mode is available in two versions. The 4Ø × 24 text mode is indicated with the command SCREEN Ø, and generally gives you 24 rows by 34 columns. You cannot use sprites on this screen. The second version of text mode is SCREEN 1, which allows you a 32 × 24 text mode. This screen appears, when you first invoke the mode, as 24 rows by 29 columns. You can use sprites on this screen.

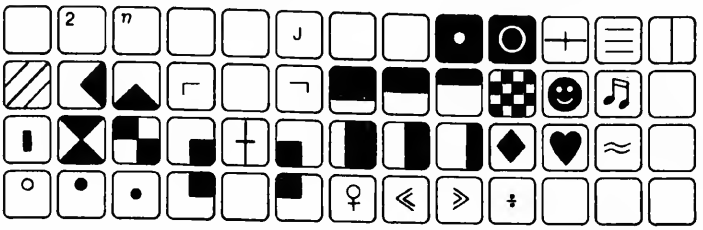
High-resolution graphics are possible in SCREEN 2 which gives you a resolution of 256 × 192 points on the screen, and allows use of sprites. You can use sprites as well in SCREEN 3, but the resolution in this mode – called the *multi-colour mode* – only allows a resolution of 64 × 48 points.

Pre-programmed Delights

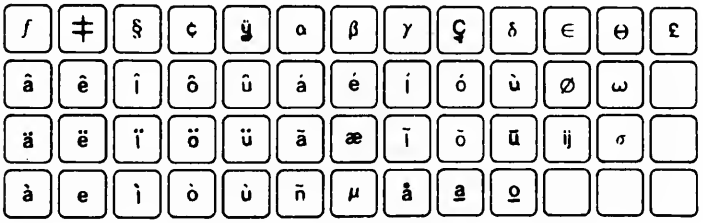
Before we go further, take note that there are a number of special graphics and foreign characters (some of which can be pressed into service as graphics) available *directly* from the keyboard, simply by pressing the GRAPH, CODE and SHIFT keys. Here's what you can get, when the GRAPH key is pressed:



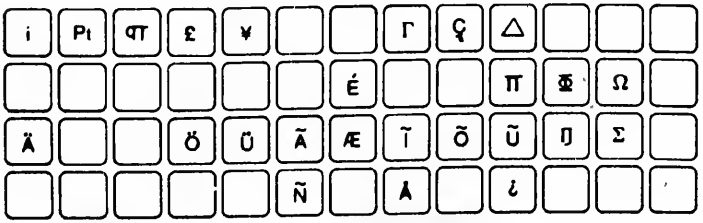
When GRAPH and SHIFT are pressed together:



When the CODE key is pressed:



When CODE and SHIFT are pressed together:



Changing colours

The COLOR statement modifies the colour of both the material printed on the screen, and the colour of the background. Run the following demonstration program which shows the relevant colours and their effects (note lines 180 and 190 which serve to put a delay in the program; they reset – in line 180 – the variable TIME to 0 and then wait – in line 190 – until its value has been incremented to 45):

```
100 REM COLOR DEMO
110 CLS:KEY OFF
120 FOR J=1 TO 15
130 FOR K=15 TO 1 STEP -1
140 LOCATE 10,10
150 IF J=K THEN 200
160 PRINT "COLOR"J","K
170 COLOR J,K
180 TIME=0
190 IF TIME<45 THEN 190
200 NEXT K
210 NEXT J
220 COLOR 1,15
```

The COLOR statement must be followed by up to three numbers. The first number controls the foreground colour, the second looks after the background and the third is the border colour. When you first turn the computer on, it is set to COLOR 15,5,4 so you can use this as a 'default' entry if you find the screen is becoming impossible to read while you are experimenting with the colours. It may well be worth assigning one of the function keys to this setting, so you can always get it back when you need it.

Here are the colours available on your MSX computer, with the numbers which trigger them:

Number	Colour
0	Transparent
1	Black
2	Medium green
3	Light green
4	Dark blue

5	Light blue
6	Dark red
7	Cyan
8	Medium red
9	Light red
10	Dark yellow
11	Light yellow
12	Dark green
13	Magenta
14	Grey
15	White

Our next program, which uses the SCREEN 2 (high resolution graphics mode) setting, shows the border colour. Line 30 and the hash 1 in line 90 is needed to be able to print on this screen:

```
10 REM COLOR DEMO SHOWING BORDER
20 SCREEN 2
30 OPEN "GRP:" FOR OUTPUT AS #1
40 CLS:KEY OFF
50 FOR J=1 TO 15
60 FOR K=15 TO 1 STEP -1
70 IF J=K THEN 130
80 CLS
90 PRINT #1,"      COLOR "J","K","J
100 COLOR J,K,J
110 TIME=0
120 IF TIME<20 THEN 120
130 NEXT K
140 NEXT J
150 COLOR 1,15
```

Getting them in Line

One of the simplest graphic control words, and the easiest to use, is the LINE statement. This allows you to draw straight lines, or rectangles, in the colour of your choice. An *optional* parameter (parameter is a word which refers to the numbers which follow a command or statement and determine how that statement or

command will act) allows you to control whether or not the rectangle is painted in.

You draw a line from the starting co-ordinates to the end ones, in a program line such as LINE (X1,Y1) – (X2,Y2) where X1 and Y1 are the co-ordinates of the starting position of the line, and X2 and Y2 are the end points. In the next program, the starting X and Y co-ordinates are called A and B and the end co-ordinates are C and D:

```
10 REM DRAW DEMO
20 DEFINT A-Z
30 SEED=RND(-TIME)
40 SCREEN 2
50 A=RND(1)*230
60 B=RND(1)*180
70 C=A+RND(1)*50
80 D=A+RND(1)*50
90 LINE (A,B)-(C,D)
100 GOTO 50
```

If we modify that program, so it reads like the following one (adding 85, and changing the end of line 90) we will get randomly-created rectangles. The "B" stands for *box,* and tells the computer to draw a rectangle, with its opposite corners located at A,B and C,D:

```
20 DEFINT A-Z
30 SEED=RND(-TIME)
40 SCREEN 2
50 A=RND(1)*230
60 B=RND(1)*180
70 C=A+RND(1)*50
80 D=A+RND(1)*50
85 COL=RND(1)*14+2
90 LINE (A,B)-(C,D),COL,B
100 GOTO 50
```

If you change line 90 so it reads as follows, you'll find the rectangles are painted in (as "BF" stands for *box fill*):

```
90 LINE (A,B)-(C,D),COL,BF
```

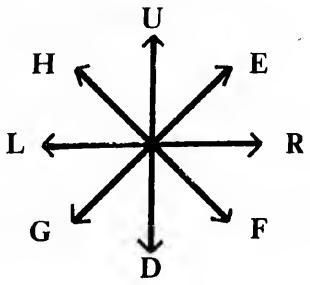
We'll be looking at your MSX computer's ability to paint or fill-in shapes in a little more detail in a moment. If you'd like sound to go with your program, modify it so it reads as follows:

```
20 DEFINT A-Z
25 SOUND 8,15:SOUND 9,15:SOUND 10,15
30 SEED=RND(-TIME)
40 SCREEN 2
50 A=RND(1)*230
60 B=RND(1)*180
70 C=A+RND(1)*50
80 D=A+RND(1)*50
85 COL=RND(1)*14+2
90 LINE (A,B)-(C,D),COL,BF
95 IF RND(1)>.6 THEN SOUND 0,A:SOUND
2,B:SOUND 4,C/2
100 GOTO 50
```

Drawing conclusions

The DRAW command is very effective on the MSX machines, and allows you to create complex shapes very, very easily. DRAW must be followed either by a string, or a string variable which has previously been assigned. The information within the string determines what the DRAW command actually does.

You move the starting point of the DRAWing you are about to create using the PSET command, and then the string takes over. The letter which *precedes* numbers within the string determines the *direction* the line will be drawn, so DRAWN "U10" will draw a line *up* ten pixels. You can draw lines up, down, right and left, as well as at the diagonals. Here's a diagram to show the letters you use to create various effects:



If you wanted to draw a specific shape, like a square, you could do it by DRAWing right, down, left and up the same number of pixels. A string to form a square could, then, be something like "R9D9L9U9" which, when preceded by DRAW, would create a square with sides nine pixels long. A diamond, by contrast, would use the E, F, G and H controls, to draw lines at the diagonals.

The next program defines three strings, to create squares, triangles and diamonds (note that *diamond* in the program is spelt without the "Ø" to avoid conflict with the MSX reserved word ON). A random starting point is chosen, and the shape, which is also randomly selected, is drawn on the screen:

```
10 REM PAINT/DRAW
20 SEED=RND(-TIME)
30 COLOR 15,1,1
40 SCREEN 2
50 TRIANGLE$="R29H15G15"
60 BOX$="R17D17L17U17"
70 DIAMND$="F12G12H12E12"
80 X=INT(RND(1)*200+12)
90 Y=INT(RND(1)*150+12)
100 PSET (X,Y)
110 CHOICE=INT(RND(1)*3)
120 IF CHOICE=0 THEN DRAW TRIANGLE$
130 IF CHOICE=1 THEN DRAW BOX$
140 IF CHOICE=2 THEN DRAW DIAMND$
150 IF RND(1)>.95 THEN CLS
160 GOTO 80
```

To produce these shapes in randomly-chosen colours, add the following line:

```
105 COLOR INT(RND(1)*14)+2,1
```

Circling Around

I pointed out a little earlier that the resolution on SCREEN 3 (the multi-colour mode) was *lower* (at 64×48 points) than the resolution

on SCREEN 2 (the high-resolution graphics mode which has 256 × 192 points). Our next program demonstrates this convincingly, using the CIRCLE command (which must be followed by the X and Y co-ordinates of the centre, and the radius measure). Run this to see a circle (an ellipse, actually) being drawn first on SCREEN 3, and then the same circle being drawn on SCREEN 2:

```
10 REM CIRCLE DEMO
20 COLOR 11,6,6
30 SEED=RND(-TIME)
40 XCRD=50+INT(RND(1)*60)
50 YCRD=50+INT(RND(1)*60)
60 RADIUS=12+INT(RND(1)*50)
70 SCREEN 3
80 CIRCLE (XCRD,YCRD),RADIUS
90 TIME=0
100 IF TIME <50 THEN 100
110 SCREEN 2
120 CIRCLE (XCRD,YCRD),RADIUS
130 TIME=0
140 IF TIME <50 THEN 140
150 GOTO 40
```

You can also use the CIRCLE command to draw arcs, by having numbers – at the end of the line including the word CIRCLE – to specify the beginning and end angles (which must be numeric expressions in radians, ranging from minus two times PI to plus two times PI).

Here's the way the command can be used for the various permutations:

CIRCLE (x co-ordinate, y co-ordinate), radius, (colour code). The colour code is optional

CIRCLE (x,y), radius, (colour code), begin angle. The colour code is optional, but you must include the relevant *comma,* even if no value is used

CIRCLE (x,y), radius, (colour code), (begin angle), end angle. Colour code and begin angle are optional, but the commas must be included

CIRCLE (x,y), radius, (colour code), (begin angle), (end angle), ratio of y radius to x radius. Colour code, begin angle and end angle are optional, but the commas must be included

The values used within the expression can be variables, rather than actual numbers. Here, for example, is a program which produces a sequence of ellipses, with progressively smaller radii:

```
10 REM DESCENDING CIRCLES
20 COLOR 1,15,15
30 SCREEN 2
40 FOR RADIUS=1 TO 95 STEP 5
50 CIRCLE (125,100),RADIUS
60 NEXT RADIUS
70 GOTO 70
```

PAINTing them in

The MSX graphics include the PAINT command which, as you've almost certainly guessed, paints in a shape drawn on the screen. Here we can see it in use with variations of the CIRCLE command, drawing circles of random size, in random positions, then painting them in.

PAINT must have, as you can see in line 110 below, the co-ordinates of its starting point, which must be *inside* the shape you want to fill in with colour. The colour used to paint the shape must be same as the colour of the shape's boundary. In this case, we've used the variable COL for the colour. Running this program, and looking closely at the effect of PAINT where the figures overlap will demonstrate more clearly than I can explain in words the effects of this command.

```
10 REM PAINT POT
20 SEED=RND(-TIME)
30 COLOR 1,15,15
40 SCREEN 2:CLS
50 X=INT(RND(1)*200)+20
60 Y=INT(RND(1)*100)+20
70 R=INT(RND(1)*40)+10
80 COL=INT(RND(1)*14)+2
90 CIRCLE(X,Y),R,COL
100 CIRCLE(X+10,Y+10),R+10,COL
110 PAINT (X,Y-9),COL
120 GOTO 50
```

Arsenic and Old Lace

Now, as we've mentioned, SCREENS 2 and 3 have different resolutions. The locations of any points on those screens can be specified by two numbers. The location Ø,Ø is the top left hand corner on both screens. The top right hand corner on SCREEN 2 is 255,Ø and its corresponding position is stated as the same on SCREEN 3, even though there are effectively far less points (or *pixels,* as they are called) on this screen. The effective resolution on SCREEN 3 is 64 × 48 (rather than the 256 × 192 on SCREEN 2). The MSX computer gets around this apparent discrepancy between the available number of pixels and the numbers used to *address* those pixels by using 16 of the SCREEN 2 pixels to make up each pixel of SCREEN 3.

Our next program will make this absolutely clear. You can turn *on* (or 'light up') any pixel with the command PSET which is followed, in brackets, by the x and y co-ordinates, separated by a comma. You turn *off* pixels by the command PRESET, again followed by the x and y co-ordinates in brackets, separated by a comma.

You can see how the MSX computer copes with the SCREENS 2 and 3 and the pixels, by running the program, which is remarkably effective (especially if you turn the colour and brightness up on your TV, and turn off the room lights):

```
10 REM AUTUMN LACE
20 COLOR 1,15,15:SCREEN 2
30 SEED=RND(-TIME)
40 ACROSS=INT(RND(1)*128)+1
50 DOWN=INT(RND(1)*96)+1
60 PSET (ACROSS,DOWN)
70 PSET (255-ACROSS,DOWN)
80 PSET (255-ACROSS,191-DOWN)
90 PSET (ACROSS,191-DOWN)
100 GOTO 40
```

Once you've run it for a while, change the 2 at the end of line 2Ø into a 3, and see how the computer handles the co-ordinates on screen 3.

Adding a little sound, and giving the computer the option of turning

pixels off as well as on, can produce an extraordinary effect, as the final program in this chapter proves convincingly:

```
10 REM STARRY, STARRY NIGHT
20 SOUND 8,15:SOUND 9,15:SOUND 10,15
30 COLOR 11,1,1:SCREEN 2
40 SEED=RND(-TIME)
50 ACROSS=INT(RND(1)*128)+1
60 DOWN=INT(RND(1)*96)+1
70 IF RND(1)>.5 THEN 90
80 SOUND 0,ACROSS+DOWN
90 IF RND(1)>.5 THEN 110
100 SOUND 2,DOWN
110 IF RND(1)>.5 THEN 130
120 SOUND 4,ABS(ACROSS-DOWN)
130 PSET (ACROSS,DOWN)
140 PSET (255-ACROSS,DOWN)
150 PSET (255-ACROSS,191-DOWN)
160 PSET (ACROSS,191-DOWN)
170 ACROSS=INT(RND(1)*128)+1
180 DOWN=INT(RND(1)*96)+1
190 PRESET (ACROSS,DOWN)
200 PRESET (255-ACROSS,DOWN)
210 PRESET (255-ACROSS,191-DOWN)
220 PRESET (ACROSS,191-DOWN)
230 IF RND(1)>.5 THEN COLOR 9,1 ELSE
IF RND(1)>.5 THEN COLOR 11,1 ELSE COL
OR 5,1
240 GOTO 50
```

Once you feel you have mastered the graphics statements in this chapter, move onto the next to discover the world of animation and sprites.

CHAPTER FOURTEEN

Animation and Sprites

Your MSX Computer is supplied with the capacity to produce *sprites,* which are graphic characters whose shape you define, and which you can easily control on the screen. They allow you to produce arcade-like moving graphics relatively simply, as this chapter will demonstrate.

Before we get on to sprites, however, I'd like to show you another simple way to get moving graphics. You'll recall in our *Duck Shoot* program that we got the 'ducks' to move across the screen by changing their positions within a string, and reprinting the string over itself, time and time again. We moved 'you' (the 'X' the first few times, and the little triangle in the final program) by reprinting you each time the program looped through. There was a blank space in the string either side of you, so if the symbol representing you was moved one place to the right or left, the blank overprinted the 'old you', so you appeared to have moved to the new position.

Locate is very useful for simple animation. All you have to do is print an object in one position, overprint that position with a blank and reprint the object in a new position, and the object will appear to move from the first position to the second. This should be very clear once you run the following program, which makes a ball bounce around the screen. As you can see from line 9Ø, the ball is a small letter "o". Enter and run the program, then return to the book for a discussion on it:

```
10 REM Bouncing Ball
20 COLOR 15,1
30 X=RND(-TIME)
40 CLS
50 A=6:B=11
```

```
60 X=1:Y=1
70 EA=A:EB=B
80 LOCATE A,B
90 PRINT "o"
100 B=B+Y
110 A=A+X
120 IF A<2 THEN X=-X:BEEP
130 IF A>35 THEN X=-X:BEEP
140 IF B<2 THEN Y=-Y:BEEP
150 IF B>18 THEN Y=-Y:BEEP
160 LOCATE EA,EB
170 PRINT " "
180 GOTO 70
```

Line 5Ø determines the starting position of the ball at A and B and line 7Ø sets two variables, EA (for 'erase A') and EB ('erase B') equal to A and B. The LOCATE statement in line 8Ø moves the cursor to position A, B and line 9Ø prints the 'ball' there. The position of the ball is updated in lines 1ØØ and 11Ø, by adding the variables X and Y to A and B. X and Y are both set initially, in line 6Ø, to equal one, but if the ball 'hits the sides' (lines 12Ø to 15Ø) the relevant control variable is changed to its minus to make the ball bounce (move back the other way). When the program gets to line 16Ø, the cursor is placed at EA, EB which – because of line 7Ø – is where the ball was printed earlier. Line 17Ø prints a blank over the ball, making it vanish. Line 18Ø sends action back to line 7Ø, where EA and EB are set to the new positions of A and B. You'll recall that A and B have been changed by lines 1ØØ and 11Ø, so EA and EB must be updated to these new positions, in order to be able to 'unprint' the old ball.

The important thing to note, when producing a program like this, is that the 'old' position of the moving object must be stored *before* a new position is defined, so the computer will know where to 'unprint' the old object.

Now, although looking at that bouncing ball, and examining the listing, can be quite instructive in terms of seeing how simple moving graphics can be produced with the aid of LOCATE, the program itself is pretty boring.

The next program, a variation of the ball one, is more interesting.

We can use the basic idea from *Bouncing Ball* for this next one, which puts you in control of a little 'bat' at the bottom of the screen. You have to keep the ball in play, by 'hitting it' back up the screen when it comes near the bat. You control your bat with the right and left arrow keys.

Enter and run the program for a while, and we'll then discuss it.

```
10 REM Bouncing ball with bat
20 DEFINT A-Z
30 COLOR 15,1
40 PLAY "O1T255L64V15"
50 PLAY "ABCDO6"
60 X=RND(-TIME)
70 SCRE=2
80 BALL=5
90 CLS
100 SOUND 13,1:SOUND 0,12
110 BAT$=CHR$(255)+CHR$(223)+CHR$(255
)
120 A=6:B=11:F=22
130 X=1:Y=1
140 EA=A:EB=B
150 LOCATE F-3,18
160 PRINT "   ";BAT$;"   "
170 A$=INKEY$
180 IF A$="" THEN 210
190 IF A$=CHR$(29) AND F>3 THEN F=F-2
200 IF A$=CHR$(28) AND F<29 THEN F=F+
2
210 LOCATE B,A
220 PRINT "O"
230 B=B+Y
240 A=A+X
250 IF A<3 THEN X=-X:PLAY "A"
260 IF A>16 THEN X=-X:PLAY "B"
270 IF B<3 THEN Y=-Y:PLAY "C"
280 IF B>33 THEN Y=-Y:PLAY "D"
290 LOCATE EB,EA
300 PRINT " "
310 IF A>16 THEN IF ABS(F-B)>3 THEN B
```

```
ALL=BALL-1:CLS:LOCATE 0,0:PRINT "Ball
 "BALL"     Score:"SCRE:PLAY"o2cdefgo3
abc.....o6":GOTO 120
320 IF BALL=0 THEN LOCATE 0,22:END
330 IF A=4 THEN SCRE=SCRE+2:LOCATE 12
,0:PRINT "Score:"SCRE
340 GOTO 140
```

The first line of interest in this program is 20. This sets all the variables used to *integer variables,* which run more quickly than non-integer, or floating-point, variables. If you do not specify a variable type, you get floating point variables. It is worth using a DEFINT A-Z line in all moving graphics games when you can, just to get maximum speed. Lines 40 and 50 produce a little music, before the screen is cleared in line 90. Line 110 sets a variable called BAT$ to equal three characters joined together. Note that when naming arrays, string and numeric variables, you can use names which are very long. However, the MSX machines only recognize the first two letters of the name (so YESTERDAY and YETI are believed to be the same variable). However, adding extra letters to the name, even if they do not help the computer, can certainly help you recognize what the various variables stand for, as is shown with BAT$ in this example.

Line 120 sets the starting points. A and B are the position of the ball, and F is the 'across' position of the bat. Line 150 moves the cursor to F-3, 18 and line 160 prints out the bat there, with two spaces either side of it. Line 170 reads the keyboard, looking for you to press the right or left arrow keys. Line 180 goes back to 170 if no key is being pressed. Line 210 moves the cursor to B, A to position the ball, and line 220 prints it. The next two lines – 230 and 240 – update the ball position, and lines 250 through to 280 check to see if the ball has hit the sides, changing the values of X or Y if it has, and sounding a note. Lines 290 and 300 erase the old ball. Line 310 checks to see if the ball is just above the line which holds the bat, and if it is, checks to see if the bat is reasonably close to the ball. If not, the ball count is cut by one, the score is updated and a small piece of music is played. Line 320 checks to see if there are any balls left, and if not, stops the program. Line 330 adds one to the score each time the ball gets near the top of the screen.

Although this is a fairly simple program, it can easily be developed to make it as elaborate as you like.

Acting Spritely

From LOCATE graphics, we can move on to the fascinating field of sprite creation on your MSX machine. We will start our investigation of sprites by writing a sprite version of the first program in this chapter, which moves a bouncing ball around the screen.

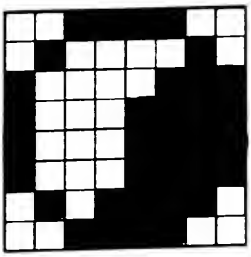
As I indicated earlier, you use the sprite screen (as opposed to the border and character pattern screens) to display and move characters which you have created yourself. Let's see, for a start, how you can create your own characters.

Each sprite is created on an 8×8 grid. Have a look at the next listing, especially lines 240 through to 310:

```
 10 REM SPRITE DEMO
 20 REM BALL BOUNCES AROUND SCREEN
 30 :
 40 SCREEN 2,0
 50 GOSUB 140:REM DEFINE SPRITE
 60 :
 70 REM PLACE SPRITE
 80 X=10:Y=10:A=10:B=9
 90 PUT SPRITE 0,(X,Y),15,0
100 X=X+A:IF X<4 OR X>247 THEN A=-A:B
EEP
110 Y=Y+B:IF Y<4 OR Y>183 THEN B=-B:B
EEP
120 GOTO 90
130 :
140 REM DEFINE SPRITE
150 S$=""
160 FOR J=1 TO 8
170 READ Z$
180 S$=S$+CHR$(VAL("&B"+Z$))
190 NEXT J
```

```
200 SPRITE$(0)=S$
210 RETURN
220 :
230 REM SPRITE DATA
240 DATA 00111100
250 DATA 01000010
260 DATA 10000111
270 DATA 10001111
280 DATA 10001111
290 DATA 10001111
300 DATA 01011110
310 DATA 00111100
```

These lines contain the data for our first sprite, which is a ball which we are going to bounce around the screen. Here is an 8 × 8 grid. On it, a square has been filled in for each 1 in the DATA statements. Each 0 indicates that the square on the grid should be left blank:



You should be able to see the shape of the ball in the filled-in squares. Even if the ball looks pretty crude to you now, you'll be pleased (and probably surprised) to see how effective it looks when you run the program. Enter the program, and then return to this book so we can explain what is going on in it.

Firstly line 40 sets the screen on which we'll be displaying our sprites, and line 50 sends the action to the subroutine from line 140 to define the sprite. It is a good idea to define your sprites at the very end of the program (perhaps from line 9000), so you can always add more if you like. Then the early part of the program can look after the actual game you're writing.

Line 80 sets the starting co-ordinates of the sprite and line 90 actually puts it in position. After the words PUT SPRITE comes the

number of the plane on which the sprite will be printed (anything from Ø to 31, with the lower numbered planes being printed *on top of* the higher numbered ones if the sprites pass over each other, as we'll see shortly). After the identifying number, we get the *location* of the sprite, in brackets. The next number determines the colour in which the sprite will be printed (white in this case) and then the final number is the *identifying number* of the sprite. The (X,Y) refers to the *top left hand corner* of the sprite, not its centre.

Lines 1ØØ and 11Ø update the position of the sprite (just as similar lines did in our LOCATE version of a bouncing ball program), change the value of A and B if the ball hits the sides, and sound a BEEP. Line 12Ø sends action back to line 9Ø to reprint the sprite in its new position. You'll see that with sprites there is no need to keep track of the 'old' position of the sprite in order to erase it. The computer does that automatically. This is one of the factors which make animation with sprites so simple.

If you want to move the sprite at a different speed, change the values assigned to A and B in line 8Ø. It is worth trying the program with them both set initially to one, just to see how smoothly sprites can move. Very large numbers will send the ball into a frenzy of action on your screen.

Once you've had enough of the bouncing ball, change lines 23Ø through to 31Ø with the following DATA statements, to get a little headless alien thing on the screen:

```
230 REM SPRITE DATA
240 DATA 00111100
250 DATA 00111100
260 DATA 01011010
270 DATA 11011011
280 DATA 10011001
290 DATA 00011000
300 DATA 01100110
310 DATA 11000111
```

From the headless alien, we will now move onto a cute little sprite which – although it's probably impossible to guess this from looking at the DATA statements – actually spells out the word "NO". Substitute these DATA lines to get the word "NO":

```
230 REM SPRITE DATA
240 DATA 00000000
250 DATA 10010111
260 DATA 11010101
270 DATA 10110101
280 DATA 10010101
290 DATA 10010101
300 DATA 10010111
310 DATA 00000000
```

Now, you probably thought, when you saw that "NO" whizzing around the screen, that it would be nice if it was a little larger, so you could see the letters more clearly. Thanks to MSX BASIC, it is very easy to double the size of sprites. Change line 40 so it reads as follows, and you'll see the word "NO" carrying on around the screen, double the size it was before:

```
40 SCREEN 2,1
```

Monsters, Monsters Everywhere

We can use the information we've acquired so far to create some pretty effective graphics games using sprites. If you enter the next program, *Monster Chase,* run it for a while, and then follow the listing through carefully, reading the relevant REM statements, you'll pick up a great deal of information which you can use in the creation of your own sprite games.

In this program, you are a yellow face. The monster is a white, evil star! You'll see there is quite a good effect when the two of you crash. The aim of this program is to keep out of the monster's clutches for as long as possible. You can use a joystick if you have one, or the arrow keys. To move diagonally using the arrow keys, hold down two at once, and you'll travel in the diagonal direction which lies between them (so holding down the upward key, and the one with the arrow pointing to the right, will move you diagonally up and to the right).

Enter and run the program, and then we'll go through it in a bit more detail:

```
10 REM Monster Chase
20 CLS:KEY OFF
30 SCREEN 2,1,0:REM
      Final 0 turns off click
40 SOUND 0,1:SOUND1,1:SOUND 8,15
50 GOSUB 630:REM define sprites
60 REM START POSITIONS
70 REM SPRITE ONE - HUMAN
80 XH=10:YH=10
90 REM SPRITE TWO - MACHINE
100 XM=159:YM=231
110 :
120 REM TURN ON SPRITE DETECTION
130 SPRITE ON
140 REM PLACE SPRITES
150 PUT SPRITE 0,(XH,YH),11,0
160 PUT SPRITE 1,(XM,YM),15,1
170 SOUND 0,(ABS(XH-XM))
180 :
190 REM SEE IF COLLIDED
200 ON SPRITE GOSUB 500
210 REM READ JOYSTICK
220 X=STICK(0)
230 IF X=1 THEN YH=YH-5
240 IF X=2 THEN YH=YH-5:XH=XH+5
250 IF X=3 THEN XH=XH+5
260 IF X=4 THEN XH=XH+5:YH=YH+5
270 IF X=5 THEN YH=YH+5
280 IF X=6 THEN YH=YH+5:XH=XH-5
290 IF X=7 THEN XH=XH-5
300 IF X=8 THEN XH=XH-5:YH=YH-5
310 IF XH<4 THEN XH=4
320 IF XH>237 THEN XH=237
330 IF YH<4 THEN YH=4
340 IF YH>183 THEN YH=183
350 :
360 REM MONSTER MOVES
370 IF XM<XH THEN XM=XM+4
380 IF RND(1)>.5 THEN 400
390 IF XM>XH THEN XM=XM-4
400 IF RND(1)>.5 THEN 420
```

```
410 IF YM<YH THEN YM=YM+4
420 IF RND(1)>.5 THEN 440
430 IF YM>YH THEN YM=YM-4
440 IF XM<4 THEN XM=4
450 IF XM>237 THEN XM=237
460 IF YM<4 THEN YM=4
470 IF YM>183 THEN YM=183
480 GOTO 150
490 :
500 REM END OF GAME
510 FOR K=1 TO 99
520 FOR Q=255 TO 0 STEP -45
530 SOUND 0,Q
540 NEXT Q
550 PUT SPRITE 0,(XH,YH),(INT(RND(1)*
16)),0
560 PUT SPRITE 1,(XH,YH),(INT(RND(1)*
16)),1
570 NEXT K
580 SOUND 8,0
590 FOR T=1 TO 1000:NEXT T
600 END
610 :
620 :
630 REM Define sprites
640 R$="":T$=""
650 FOR J=1 TO 8
660 READ A$
670 R$=R$+CHR$(VAL("&B"+A$))
680 NEXT J
690 FOR J=1 TO 8
700 READ A$
710 T$=T$+CHR$(VAL("&B"+A$))
720 NEXT J
730 SPRITE$(0)=R$
740 SPRITE$(1)=T$
750 RETURN
760 :
770 REM SPRITE DATA
780 REM  - ONE -
790 DATA 00011110
```

```
800 DATA 00111111
810 DATA 00101111
820 DATA 01111111
830 DATA 00110111
840 DATA 00000111
850 DATA 00111110
860 DATA 00011110
870 REM  - TWO -
880 DATA 10010010
890 DATA 01010100
900 DATA 00111000
910 DATA 11111110
920 DATA 00111000
930 DATA 01010100
940 DATA 10010010
950 DATA 00000000
```

Line 130, as the REM statement in line 120 points out, turns on the mechanism in the computer which detects if two sprites have collided. If they have, line 200 (using ON SPRITE) goes to the subroutine at line 500 which ends the game. Lines 150 and 160 place you and the monster on the screen. You are sprite zero in yellow and the computer is the white sprite. Lines 220 through to 300 read the joystick (or arrow keys) and use this reading to modify the position of your sprite. Lines 310 through to 340 keep you on the screen.

The 'intelligence' of the MSX Monster is determined by lines 370 through to 470. The RND(1) bits, which get it to jump over some moves, ensure that the monster does not just head straight for you, and end the game in a matter of seconds. You can see, by looking at the ends of lines 370, 390, 410 and 430, that the monster moves only four pixels at a time (compared to your five) in order to allow you some hope of reasonable survival.

Line 480 sends action back to 150 which continues to cycle through until the ON SPRITE line (200) detects a collision between the two of you, and sends you to the end of game routine. At this routine, the K and Q loops print the two sprites on top of each other in randomly-chosen colours, complete with a violent flurry of sound. Line 590 holds the program for a while, before it ends on line 600, and drops out of SCREEN 2.

A final point. Look at the Ø at the end of line 3Ø. This turns off the 'key click' sound which otherwise, using the joystick-reading routine, would drive you crazy with constant clicking.

Once you've mastered the first version of this game, and you understand more or less what all the bits in it do, you can enter our next sprite program. This is a modification of the previous one, so you need only *add* new lines to the program you already have in your computer (just enter all the lines which have numbers not ending in zero plus those at the end) to produce the second game. This version of the program adds a second monster, a red bomb-shaped thing. You have to avoid both of them. Note that you can be very clever, and try and make the two monsters run into each other. If you can do this, before one of them gets to you, you'll be the winner.

```
10 REM Monster Chase
20 CLS:KEY OFF
30 SCREEN 2,1,0:REM
      Final Ø turns off click
40 SOUND Ø,1:SOUND1,1:SOUND 8,15
50 GOSUB 630:REM define sprites
60 REM START POSITIONS
70 REM SPRITE ONE - HUMAN
80 XH=10:YH=10
90 REM SPRITE TWO - MACHINE
100 XM=159:YM=231
101 REM SPRITE THREE - MACHINE
102 X2=110+INT(RND(1)*40):Y2=33+INT(R
ND(1)*30)
110 :
120 REM TURN ON SPRITE DETECTION
130 SPRITE ON
140 REM PLACE SPRITES
150 PUT SPRITE Ø,(XH,YH),11,Ø
160 PUT SPRITE 1,(XM,YM),15,1
165 PUT SPRITE 2,(X2,Y2),9,2
170 SOUND Ø,(ABS(XH-XM))
180 :
190 REM SEE IF COLLIDED
200 ON SPRITE GOSUB 500
210 REM READ JOYSTICK
```

```
220 X=STICK(0)
230 IF X=1 THEN YH=YH-5
240 IF X=2 THEN YH=YH-5:XH=XH+5
250 IF X=3 THEN XH=XH+5
260 IF X=4 THEN XH=XH+5:YH=YH+5
270 IF X=5 THEN YH=YH+5
280 IF X=6 THEN YH=YH+5:XH=XH-5
290 IF X=7 THEN XH=XH-5
300 IF X=8 THEN XH=XH-5:YH=YH-5
310 IF XH<4 THEN XH=4
320 IF XH>237 THEN XH=237
330 IF YH<4 THEN YH=4
340 IF YH>183 THEN YH=183
350 :
360 REM MONSTER MOVES
370 IF XM<XH THEN XM=XM+7
380 IF RND(1)>.5 THEN 400
390 IF XM>XH THEN XM=XM-7
400 IF RND(1)>.5 THEN 420
410 IF YM<YH THEN YM=YM+4
420 IF RND(1)>.5 THEN 440
430 IF YM>YH THEN YM=YM-4
440 IF XM<4 THEN XM=4
450 IF XM>237 THEN XM=237
460 IF YM<4 THEN YM=4
470 IF YM>183 THEN YM=183
472 IF X2<XH THEN X2=X2+3
474 IF RND(1)>.5 THEN 478
476 IF X2>XH THEN X2=X2-5
478 IF RND(1)>.5 THEN 482
480 IF Y2<YH THEN Y2=Y2+4
482 IF RND(1)>.5 THEN 499
484 IF Y2>YH THEN Y2=Y2-5
490 :
499 GOTO 150
500 REM END OF GAME
510 FOR K=1 TO 99
520 FOR Q=255 TO 0 STEP -45
530 SOUND 0,Q
540 NEXT Q
550 PUT SPRITE 0,(XH,YH),(INT(RND(1)*
```

```
16)),0
560 PUT SPRITE 1,(XH,YH),(INT(RND(1)*
16)),1
565 PUT SPRITE 2,(XH,YH),(INT(RND(1)*
16)),2
570 NEXT K
580 SOUND 8,0
590 FOR T=1 TO 1000:NEXT T
600 END
610 :
620 :
630 REM Define sprites
640 R$="":T$=""
650 FOR J=1 TO 8
660 READ A$
670 R$=R$+CHR$(VAL("&B"+A$))
680 NEXT J
690 FOR J=1 TO 8
700 READ A$
710 T$=T$+CHR$(VAL("&B"+A$))
720 NEXT J
721 SEED=RND(-TIME):F$="":FOR J=1 TO
8
722 READ A$
723 F$=F$+CHR$(VAL("&B"+A$))
724 NEXT J
730 SPRITE$(0)=R$
740 SPRITE$(1)=T$
745 SPRITE$(2)=F$
750 RETURN
760 :
770 REM SPRITE DATA
780 REM  - ONE -
790 DATA 00011110
800 DATA 00111111
810 DATA 00101111
820 DATA 01111111
830 DATA 00110111
840 DATA 00000111
850 DATA 00111110
860 DATA 00011110
```

```
870 REM  - TWO -
880 DATA 10010010
890 DATA 01010100
900 DATA 00111000
910 DATA 11111110
920 DATA 00111000
930 DATA 01010100
940 DATA 10010010
950 DATA 00000000
960 REM - THREE -
970 DATA 01101100
980 DATA 00101000
990 DATA 00010000
1000 DATA 00111000
1010 DATA 01111100
1020 DATA 00111000
1030 DATA 00010000
1040 DATA 00000000
```

You may well wish to continue to experiment with this game, adding new monsters, or simply redefining the ones you already have.

Doing it in Multiples

From the relative quiet of *Monster Chase* we turn to *Multi-Sprite Zap Out* which, if nothing else, probably wins an award for the worst computer game title of the decade. In this program, there are six sprites. One is under your control, and the rest zap around the screen, apparently at random. Whereas in the *Monster Chase* games you had to avoid the computer sprites for as long as possible, in this game you try to hit as many of them as you can. Each collision (and collisions between monsters are worth as much as a head-on smash between you and a monster) adds to your score. The program continues until the variable TIME, which is set to zero at the beginning of the run, equals 1000. At this point, you'll be told your score. This is then compared to the high score, which is updated if necessary. You'll find you spend a lot of time on this program, trying to clock up ever-higher scores.

Here's the listing of *Multi-Sprite Zap Out:*

```
10 REM MULTI-SPRITE ZAP OUT
20 DEFINT A-Z
30 OLD=4
40 HISC=0
50 SEED=RND(-TIME)
60 RESTORE
70 TIME=0
80 COUNT=0
90 SCREEN 2,1,0
100 GOSUB 1180:REM DEFINE SPRITES
110 :
120 REM PLACE SPRITES
130 SOUND 8,15
140 HX=12:HY=11
150 X1=42:Y1=11
160 X2=92:Y2=95
170 X3=156:Y3=112
180 X4=12:Y4=100
190 X5=99:Y5=155
200 SPRITE ON
210 :
220 PUT SPRITE 0,(HX,HY),1,0
230 PUT SPRITE 1,(X1,X1),9,1
240 PUT SPRITE 2,(X2,Y2),11,2
250 PUT SPRITE 3,(X3,Y3),3,3
260 PUT SPRITE 4,(X4,Y4),6,4
270 PUT SPRITE 5,(X5,Y5),7,5
280 :
290 IF TIME>1000 THEN 910
300 K=TIME/4
310 FLAG=0
320 IF RND(1)>.95 THEN A=A+1:B=B+1:C=
C+1:D=D+1
330 ON SPRITE GOSUB 860
340 IF FLAG THEN SOUND 8,15:SOUND 0,K
:SOUND 0,K+3:SOUND 0,K-3
350 SOUND 8,0
360 :
370 REM MOVE PLAYER
380 X=STICK(0)
390 IF X=0 THEN X=OLD
```

```
400 IF X=1 THEN HY=HY-9
410 IF X=2 THEN HY=HY-9:HX=HX+9
420 IF X=3 THEN HX=HX+9
430 IF X=4 THEN HX=HX+9:HY=HY+9
440 IF X=5 THEN HY=HY+9
450 IF X=6 THEN HY=HY+9:HX=HX-9
460 IF X=7 THEN HX=HX-9
470 IF X=8 THEN HX=HX-9:HY=HY-9
480 OLD=X
490 IF HX<4 THEN HX=227
500 IF HX>227 THEN HX=4
510 IF HY<4 THEN HY=183
520 IF HY>183 THEN HY=4
530 :
540 REM MOVE MSX THINGS
550 X1=X1+A:Y1=Y1-2
560 IF X1>227 THEN X1=4
570 IF Y1>183 THEN Y1=4
580 IF X1<4 THEN X1=227
590 IF Y1<4 THEN Y1=183
600 :
610 X2=X2+B:Y2=Y2-2
620 IF X2>227 THEN X2=4
630 IF Y2>183 THEN Y2=4
640 IF X2<4 THEN X2=227
650 IF Y2<4 THEN Y2=183
660 :
670 X3=X3-C:Y3=Y3-7
680 IF X3>227 THEN X3=4
690 IF Y3>183 THEN Y3=4
700 IF X3<4 THEN X3=227
710 IF Y3<4 THEN Y3=183
720 :
730 X4=X4+D:Y4=Y4-3
740 IF X4>227 THEN X4=4
750 IF Y4>183 THEN Y4=4
760 IF X4<4 THEN X4=227
770 IF Y4<4 THEN Y4=183
780 :
790 X5=X5+A:Y5=Y5-D
800 IF X5>227 THEN X5=4
```

```
810 IF Y5>183 THEN Y5=4
820 IF X5<4 THEN X5=227
830 IF Y5<4 THEN Y5=183
840 GOTO 220
850 :
860 REM COLLISION COUNT
870 COUNT=COUNT+1
880 FLAG=1
890 RETURN
900 :
910 REM END OF GAME
920 SCREEN 0
930 FOR T=0 TO 15
940 COLOR T,15-T:FOR J=1 TO 40:NEXT J
950 NEXT T
960 COLOR 6,11
970 SOUND 8,0
980 LOCATE 14,6
990 SOUND 8,0
1000 PRINT "TIME IS UP"
1010 PRINT
1020 PRINT TAB(11);"YOU SCORED"COUNT
1030 IF COUNT>HISC THEN HISC=COUNT
1040 PRINT
1050 PRINT TAB(9);"THE HI-SCORE IS"HI
SC
1060 K=RND(1)*10+4:SOUND 8,15
1070 FOR T=0 TO 15
1080 COLOR T,15-T:SOUND 1,T
1090 FOR J=1 TO 270:NEXT J
1100 NEXT T
1110 FOR J=0 TO 255 STEP K
1120 SOUND 0,J:FOR M=1 TO 10:NEXT M
1130 NEXT J
1140 COLOR 1,15
1150 GOTO 60
1160 END
1170 :
1180 REM INITIALISE
1190 A=RND(1)*7+1-RND(1)*7+1
1200 B=RND(1)*7+1-RND(1)*7+1
```

```
1210 C=RND(1)*7+1-RND(1)*7+1
1220 D=RND(1)*7+2-RND(1)*7+1
1230 :
1240 REM DEFINE SPRITES
1250 Q$="":T$="":R$="":U$=""
1260 FOR J=1 TO 8
1270 READ Z$
1280 Q$=Q$+CHR$(VAL("&B"+Z$))
1290 NEXT J
1300 SPRITE$(0)=Q$
1310 :
1320 FOR J=1 TO 8
1330 READ Z$
1340 T$=T$+CHR$(VAL("&B"+Z$))
1350 NEXT J
1360 SPRITE$(1)=T$
1370 SPRITE$(4)=T$
1380 :
1390 FOR J=1 TO 8
1400 READ Z$
1410 R$=R$+CHR$(VAL("&B"+Z$))
1420 NEXT J
1430 SPRITE$(2)=R$
1440 :
1450 FOR J=1 TO 8
1460 READ Z$
1470 U$=U$+CHR$(VAL("&B"+Z$))
1480 NEXT J
1490 SPRITE$(3)=U$
1500 SPRITE$(5)=U$
1510 RETURN
1520 :
1530 REM SPRITE DATA
1540 DATA 10111101
1550 DATA 10011001
1560 DATA 10100101
1570 DATA 11000011
1580 DATA 11000011
1590 DATA 10100101
1600 DATA 10011001
1610 DATA 01111110
```

```
1620 :
1630 DATA 00111100
1640 DATA 01011010
1650 DATA 11011011
1660 DATA 10111101
1670 DATA 01100110
1680 DATA 00100100
1690 DATA 00100100
1700 DATA 01100110
1710 :
1720 DATA 11000011
1730 DATA 01100110
1740 DATA 00100100
1750 DATA 00111100
1760 DATA 01100110
1770 DATA 01100110
1780 DATA 00111100
1790 DATA 00011000
1800 :
1810 DATA 10011001
1820 DATA 11011011
1830 DATA 01011010
1840 DATA 10011001
1850 DATA 11111111
1860 DATA 00100100
1870 DATA 01000010
1880 DATA 11000011
```

As an exercise, you can now modify all the sprite shapes used in the last program. Then, you can work at changing the direction and speed at which they move. Finally, when you think you've got the whole thing licked, you can try your hand at producing sprites which occupy a 16 × 16 grid, rather than the 8 × 8 one we've been using. If you select SCREEN 2,2,Ø, and use 32 DATA statements (the first 16 look after the left hand side of your sprite, and the second 16 control the appearance of the right hand side), you'll be able to create very dramatic 16 × 16 sprites. Change the SCREEN line to SCREEN 2,3,Ø and the sprites will appear on the screen in double their former size. Then you'll really be creating something exciting.

As a final note, we should mention *interrupts* which you can use to control sprites and other elements of moving graphics programs. In

the programs in this chapter, we've put the control of the sprites in a loop. Each time a particular line is reached, the position of the sprite is changed.

If you would rather have the sprites moving regularly, regardless of what is happening elsewhere in the program, or you want something else to happen on a regular basis (such as BEEP sounding) you can use interrupts.

Enter this simple program:

```
10 REM INTERRUPTS
20 STRIG(0) ON
30 ON STRIG GOSUB 50
40 GOTO 30
50 PRINT "THIS IS SUBROUTINE"
60 BEEP:BEEP
70 RETURN
```

The STRIG(Ø) ON line (2Ø) tells the computer to turn on the interrupt detector routine which, in line 3Ø, will go to the subroutine in line 5Ø whenever the space bar is pressed. If you want to check if the fire button on your joystick has been pressed, change the zero in brackets in line 2Ø into a 1. This checks the fire button on the joystick in socket one. A 2 in line 2Ø will check the fire button on joystick two. If your joysticks have a second fire button each, these can be checked wtih a 3 line 2Ø (second button on joystick in socket one) or a 4 (second button on the joystick in the second socket).

Interrupts are extremely useful parts of the MSX vocabulary. The command ON INTERVAL can be used to do something at specific times, regardless of what else is happening. The following program, for example, will beep every time the TIME counter gets to 15Ø:

```
10 REM On Interval Demo
20 CLS:KEY OFF
30 ON INTERVAL=150 GOSUB 110
40 TIME=0
50 INTERVAL ON
60 J=0
70 J=J+1
```

```
80 LOCATE 5,5
90 PRINT J
100 GOTO 70
110 PLAY "A"
120 LOCATE 5,5
130 PRINT "THIS IS IT"J
140 FOR K=1 TO 500:NEXT K
150 LOCATE 5,5
160 PRINT "              "
170 LOCATE 5,5
180 TIME=0
190 RETURN
```

The ON KEY GOSUB line reads the function keys, and does specific things when they are pressed, as you'll see by first running the following program, and then examining the listing:

```
100 REM On Key Demo
110 CLS:KEY OFF
120 PLAY "T64 L64"
130 KEY (1) ON
140 KEY (2) ON
150 KEY (3) ON
160 KEY (4) ON
170 KEY (5) ON
180 ON KEY GOSUB 200,230,260,290,320
190 GOTO 190
200 PLAY "C"
210 LOCATE 5,5:PRINT "KEY ONE  "
220 RETURN
230 PLAY "D"
240 LOCATE 5,5:PRINT "KEY TWO  "
250 RETURN
260 PLAY "E"
270 LOCATE 5,5:PRINT "KEY THREE"
280 RETURN
290 PLAY "F"
300 LOCATE 5,5:PRINT "KEY FOUR "
310 RETURN
320 PLAY "G"
330 LOCATE 5,5:PRINT "KEY FIVE "
340 RETURN
```

On the simple 'piano' created by the function keys, you can play a surprising variety of tunes.

Finally, you can use ON STOP GOSUB to automatically send the program to, for example, a subroutine at the end of the listing which clears the screen and lists the program. Then, when you're developing a program, you can get a listing automatically up on the screen in the middle of a run, just by pressing the CTRL and STOP keys.

CHAPTER FIFTEEN

MSX Checkers

Time for another break in our learning. In this chapter, you come face to face with the intelligent might of the MSX electronic brain, as it challenges you to a round of Checkers, or Draughts as the game is generally known in the UK.

The game of Checkers has a long and honourable history. R. C. Bell, in his book *Discovering Old Board Games* (Shire Publications, Aylesbury, UK, 198Ø) says it was invented around 11ØØ "probably in the south of France, using Backgammon tablemen on a chequered chessboard with the Alquerque method of capture" (pp. 35-36). The *Encyclopedia of Sports, Games and Pastimes* (Fleetway House, London, c. 1935) puts it much further back in time: "Forms of it were known in ancient Egypt, Greece and Rome, while the game was played in the mid-seventeenth century much as it is today" (p. 237).

Regardless of its age, it is a very popular game around the world, with many European countries having regional variations on the game of their own. Continental draughts, for example, is played on a board of 1ØØ squares, with each player starting the game with 2Ø pieces. It was developed in the early 17ØØs.

This *MSX Checkers* plays the game you are probably most familiar with. It plays swiftly, and reasonably well, although its lack of endgame strategy often leads to a dramatic collapse in the final moments of a game.

When the program begins, you're at the bottom of the screen, playing up and the computer is at the top playing down. The program – as it stands – gives the computer the first move. If you want the first move, delete line 3Ø. Your kings are shown as K's, with

the MSX machine's kings as dollars signs. You'll find the program will lead you through the entry of your moves, and will respond swiftly to your game, rarely making a mistake as its silicon brain sifts through the many thousands of options open to it at any one time.

```
10 REM MSX Checkers
20 GOSUB 1170
30 GOTO 70
40 :
50 GOSUB 640
60 GOSUB 900
70 GOSUB 640
80 GOSUB 110
90 GOTO 50
100 :
110 FOR X=1 TO 10:S(X)=0:NEXT X
120 SC=0:A=89
130 A=A-1
140 IF Q(A)<>C AND Q(A)<>CK THEN 250
150 B=0:IF A<29 THEN B=2
160 B=B+1
170 M=A+N(B)
180 IF M>88 OR M<11 THEN 250
190 IF (Q(M)=H OR Q(M)=HK) AND Q(M+N(
B))=E THEN 310
200 IF Q(M)<>E THEN 240
210 IF NOT (Q(M-11)<>H AND Q(M-11)<>H
K) THEN 240
220 IF NOT (Q(M-9)<>H AND Q(M-9)<>HK)
 AND Q(M+9)<>HK THEN 240
230 IF ((Q(M+22)<>HK OR Q(M+18)<>HK)
AND (Q(M+9)<>C OR Q(M+9)<>CK OR Q(M+1
1)=C OR Q(M+11)=CK)) AND Q(M+11)<>HK
THEN GOSUB 430
240 IF B<2 OR (Q(A)=CK AND B<4) THEN
160
250 IF A>11 THEN 130
260 :
270 FL=0
280 IF Q(22)=C OR Q(24)=C OR Q(26)=C
```

```
OR Q(28)=C THEN GOSUB 1400
290 IF FL=1 THEN 610
300 GOTO 450
310 Q(M+N(B))=Q(A):Q(M)=E:Q(A)=E
320 CM=CM+1:GOSUB 640
330 A=M+N(B)
340 B=0
350 B=B+1
360 IF (A+2*M(B)<11 OR A+2*N(B)>88) A
ND B<4 THEN 350
370 M=A+N(B)
380 IF Q(M)=C AND B>2 THEN RETURN
390 IF M+N(B)>10 AND M+N(B)<89 THEN I
F (Q(M)=H OR Q(M)=HK) AND Q(M+N(B))=E
 THEN 310
400 IF B<2 OR (Q(A)=CK AND B<4) THEN
350
410 RETURN
420 :
430 IF SC<10 THEN SC=SC+1
440 S(SC)=100*A+B+20:RETURN
450 IF SC=0 THEN 510
460 XC=RND(1)*SC+1
470 A=S(XC)/100
480 M=A+N(S(XC)-100*A-20)
490 GOTO 610
500 :
510 SC=SC+1:A=RND(1)*88+1
520 IF Q(A)<>C AND Q(A)<>CK THEN 590
530 B=0
540 B=B+1
550 M=A+N(B)
560 IF M>88 OR M<11 THEN 580
570 IF Q(M)=E THEN 610
580 IF B<2 OR Q(A)=CK AND B<4 THEN 54
0
590 IF SC<300 THEN 510
600 PRINT:PRINT "I concede the game":
END
610 Q(M)=Q(A):Q(A)=E
620 RETURN
```

```
630 :
640 LOCATE 0,3
650 PLAY "CEDF"
660 PRINT TAB(12);"MSX Computer:"CM
670 PRINT TAB(15);"Human:"HU
680 PRINT
690 PRINT TAB(12);"1 2 3 4 5 6 7 8"
700 PRINT TAB(12);"---------------"
710 FOR F=80 TO 10 STEP -10
720 PRINT TAB(8);F/10;"¦";
730 FOR G=1 TO 8
740 PRINT CHR$(Q(F+G));
750 IF G<8 THEN PRINT " ";
760 NEXT G
770 PRINT "¦";F/10
780 NEXT F
790 PRINT TAB(12);"---------------"
800 PRINT TAB(12);"1 2 3 4 5 6 7 8"
810 IF CM=12 OR HU=12 THEN 830
820 RETURN
830 PRINT
840 IF HU=12 THEN PRINT TAB(8);"You h
ave won"
850 IF CM=12 THEN PRINT TAB(10);"I ha
ve won"
860 PRINT TAB(8);"Thanks for the game
"
870 END
880 :
890 REM   Enter 99 to concede
900 LOCATE 8,22
910 PRINT "Enter your move: From";
920 INPUT A
930 IF A=99 THEN 860
940 IF Q(A)<>H AND Q(A)<>HK THEN 920
950 LOCATE 8,22
960 PRINT "
"
970 LOCATE 8,22
980 PRINT "              From"A"to";
990 INPUT B
```

```
1000 LOCATE 8,22
1010 PRINT "
  "
1020 IF Q(B)<>E THEN 950
1030 Q(B)=Q(A):Q(A)=E
1040 :
1050 FOR T=11 TO 17:IF Q(T)=C THEN Q(
T)=CK
1060 NEXT T
1070 FOR T=82 TO 88:IF Q(T)=H THEN Q(
T)=HK
1080 NEXT T
1090 :
1100 IF ABS(A-B)<12 THEN RETURN
1110 HU=HU+1:Q((A+B)/2)=E:GOSUB 640
1120 LOCATE 0,22:INPUT "Can you jump
again (Y or N)";Z$
1130 LOCATE 0,22:PRINT "
                        "
1140 IF Z$<>"y" AND Z$<>"Y" THEN RETU
RN
1150 A=B:GOTO 940
1160 :
1170 REM ** Initialise **
1180 X=RND(-TIME)
1190 DEFINT A-Z
1200 COLOR 15,1
1210 PLAY "T255 L64 O2"
1220 CLS:LOCATE 9,10
1230 PRINT "Please stand by..."
1240 DIM Q(99),N(4),S(10)
1250 H=ASC("H"):HK=ASC("K")
1260 C=ASC("C"):CK=ASC("$")
1270 E=32:OF=-99
1280 FOR M=1 TO 99:Q(M)=OF:NEXT M
1290 FOR M=1 TO 64
1300 READ D,G
1310 Q(D)=G
1320 NEXT M
1330 FOR M=1 TO 4
1340 READ X:N(M)=X
```

```
1350 NEXT M
1360 CM=0:HU=0
1370 CLS
1380 RETURN
1390 :
1400 IF Q(22)=C AND Q(11)=E THEN A=22
:M=11:FL=1:RETURN
1410 IF Q(22)=C AND Q(13)=E THEN A=22
:M=13:FL=1:RETURN
1420 IF Q(24)=C AND Q(13)=E THEN A=24
:M=13:FL=1:RETURN
1430 IF Q(24)=C AND Q(15)=E THEN A=24
:M=15:FL=1:RETURN
1440 IF Q(26)=C AND Q(15)=E THEN A=26
:M=15:FL=1:RETURN
1450 IF Q(26)=C AND Q(17)=E THEN A=26
:M=17:FL=1:RETURN
1460 RETURN
1470 :
1480 DATA 81,255,82,67,83,255,84,67,8
5,255,86,67,87,255
1490 DATA 88,67,71,67,72,255,73,67,74
,255,75,67,76,255
1500 DATA 77,67,78,255,61,255,62,67,6
3,255,64,67
1510 DATA 65,255,66,67,67,255,68,67,5
1,32,52,255
1520 DATA 53,32,54,255,55,32,56,255,5
7,32,58,255
1530 DATA 41,255,42,32,43,255,44,32,4
5,255,46,32
1540 DATA 47,255,48,32,31,72,32,255,3
3,72,34,255,35,72
1550 DATA 36,255,37,72,38,255,21,255,
22,72,23,255,24,72
1560 DATA 25,255,26,72,27,255,28,72,1
1,72,12,255,13,72
1570 DATA 14,255,15,72,16,255,17,72,1
8,255
1580 DATA -11,-9,11,9
```

CHAPTER SIXTEEN

Creating and Playing Adventures

Let's face it. Life can be pretty tame, sometimes. There don't seem to be many dragons waiting to be slain in my city, and chests heaped high with abandoned gold are in scarce supply. I can't remember the last time I met an Evil Magician down at the local supermarket, and it's been ages since I discussed battle tactics with sentient androids at the local tavern.

The hunger for excitement lies in all of us. The desire to take on the personalities of other, more vibrant, people – even for just an hour or so – is a common one. Although you can't conjure up devils and werebears, invoke the power of a Shield of Protection or employ trolls to carry sacks of emeralds from the ruins of an abandoned castle, role-playing games allow you to do just that.

Adventure gaming has hit the big time. You've probably seen the claims that it is the 'fastest growing game in the world'. Whether that's true or not, it indicates that adventure gaming is a leisure pursuit which satisfies the inner needs of many people.

You may well have taken part in adventure role-playing games yourself.

But these real-life campaigns have one enormous disadvantage. You need people to play with and against. You need a referee (often called the Game Master, or Dungeon Master) to control the world and its artifacts and encounters. It is not always particularly easy to get all these other human beings together just when you decide you'd like to indulge in a little bit of adventuring. That's where the computer can come in.

Although MSX computer adventure games lack a little of the spontaneity of such games when played with live company, they can be remarkably unpredictable and exciting to play. The fact that the dreaded *MSX Monster* exists only within the computer's RAM seems in no way to diminish the relief you feel when it dies. The gems you find lying all over the place are no less 'real' than those discovered in live-action adventures.

The word adventure, then, is used to describe the class of computer games in which the player moves through an alternative reality. In this "otherworld" there are monsters to be fought, treasures to be discovered, maps to be made, and puzzles to be solved.

One feature of true adventure games is that the reality they model is *consistent.* That is, the world created within the adventure program is solid, and – apart from any events specific to that game, such as an earthquake, or a magic spell – the parts of the world do not shift in a random fashion. In a properly-constructed adventure the rivers stay in place, dungeon walls do not mysteriously shift and move every time you turn your back, and objects you put down in one cave within an underground labyrinth do not suddenly appear of their own volition a hundred leagues away.

Map-making is one of the true adventure-player's skills and delights. Working your way through an imaginary, but self-consistent environment, tackling monsters and collecting treasures, solving puzzles as you go up and down staircases and chutes, exploring side tunnels, getting lost in self-circling mazes and so on, is only fascinating if the world you are exploring is mappable. The "worlds" created in this chapter of the book can, of course, be mapped.

Mapping the environment

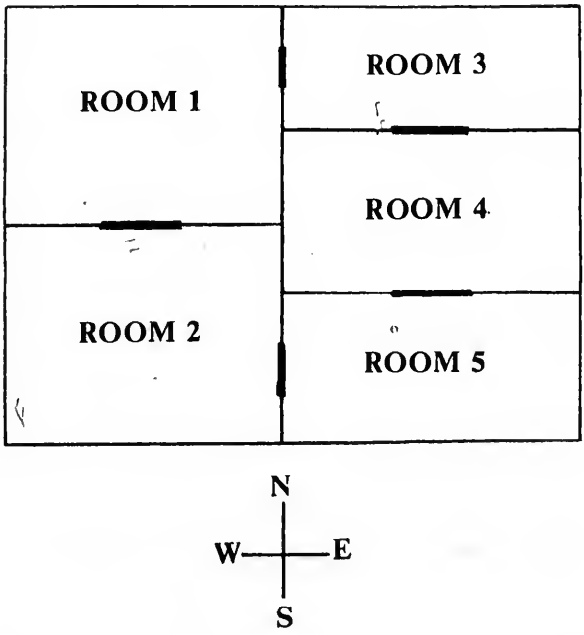
An adventure environment must be coherent. That is, the explorer making his way through the environment must be able to draw up a complete map as he works his way through it. If he draws a door connecting the study with the library on an environment floor plan, because he has discovered that going through the study door leads into the library, he is entitled to expect that turning around and

going back will bring him back into the study. The game-player should be able to build up an entire plan in this way, checking his plan from time to time by 'walking around' the house, castle, forest, underground labyrinth or whatever where the adventure is taking place.

The first step, then, in building an adventure program is to construct an environment which can be both mapped, and represented in some way which the computer can store.

You'll be pleased to know it is relatively easy to satisfy both these conditions.

Look at the following five-room environment, a very simple one, which we shall treat as though it was a computer adventure environment.



The key to holding an environment like this in a way the computer can understand and manipulate is to set up an array, each element of which represents a room. The solid markers between rooms are doors.

If you were in room one, you could move east into room three, or south into room two. In room four you can move north into room three and south into room five, and so on. Imagine we have set up an array, which we have dimensioned as DIM A(5,4). The first dimension is the room, and the second one is the four possible directions from that room (that is, north, south, east and west).

Armed with the map of the five-room environment, we can now build up a *travel table*, which can then be fed into the array, to allow us to move from point to point within the environment. Here's the travel table for the simple, five-roomed environment we've mapped:

ROOM	N	S	E	W
1	Ø	2	3	Ø
2	1	Ø	5	Ø
3	Ø	4	Ø	1
4	3	5	Ø	Ø
5	4	Ø	Ø	2

Take some time to study this table, and the way it relates to the map, because it is the single most important key to building adventure programs you can learn.

Look at the table for room one. Under the 'N' (for north) column, we see a zero, meaning you cannot move north from room one (a fact which is easily verified by looking at our map). However, under the "S" we see the number two, meaning that if we travelled south from room one we would end up in room two (again you can verify this from the map). Move east (the 'E' column) from room one, and you'll end up in room three. The Ø in the 'W' column means there is no travel possible west from room one.

You can work right through the table, if you like, checking that the numbers on it correspond to the 'reality' of the map.

Now, to allow the player to move around the environment, we only need to (a) fill each element of the array with the relevant information from the *travel table;* (b) tell the player where he or she is; and (c) allow the decisions entered by the player regarding the direction he or she wants to move to be checked against the array, and then – if possible – updated to reflect the player's new location.

It is easier to do this than you might think.

Moving about

Firstly, we need to write a small program to feed the relevant information into the array. Two simple READ/DATA loops like the following will do it:

```
100 DIM A(5,4)
110 FOR B=1 TO 5
120 FOR C=1 TO 4
130 READ A(B,C)
140 NEXT C
150 NEXT B
160 :
170 DATA 0,2,3,0
180 DATA 1,0,5,0
190 DATA 0,4,0,1
200 DATA 3,5,0,0
210 DATA 4,0,0,2
```

As you can see, the DATA statements correspond exactly with the items in our Travel Table.

The player's location

If we decided that the room the player is currently occupying could be designated by the variable RO we could tell the player where he or she was as follows, as well as indicating which exits existed:

```
100 PRINT "YOU ARE NOW IN ROOM NUMBER
"RO
110 IF A(RO,1)<>0 THEN PRINT "A DOOR
LEADS NORTH"
120 IF A(RO,2)<>0 THEN PRINT "THERE I
S AN EXIT TO THE SOUTH"
130 IF A(RO,3)<>0 THEN PRINT "YOU CAN
 LEAVE VIA THE EAST"
140 IF A(RO,4)<>0 THEN PRINT "A DOORW
AY OPENS TO THE WEST"
```

The player's input could be a single letter ("N" for north, and so on) and the program could look at the input, and check to see if an exit to that direction existed:

```
650 INPUT "WHICH WAY DO YOU WANT TO G
O";MO$
660 :
670 :
680 :
690 IF MO$="N" AND A(RO,1)=0 THEN PRI
NT "NO EXIT THAT WAY":GOTO 650
700 IF MO$="S" AND A(RO,2)=0 THEN PRI
NT "THERE IS NO EXIT SOUTH":GOTO 650
710 IF MO$="E" AND A(RO,3)=0 THEN PRI
NT "YOU CANNOT GO IN THAT DIRECTION":
GOTO 650
720 IF MO$="W" AND A(RO,4)=0 THEN PRI
NT "YOU CANNOT MOVE THROUGH STONE":GO
TO 650
730 :
740 :
750 :
```

Consistency and reality

Although the rooms only exist on paper and in elements in an array, the fact that they behave like 'real rooms' soon allows them to be perceived as though they were solid and real in a way which is uncanny. Add descriptions of each room – YOU ARE IN A SMALL WORKMAN'S HUT ATTACHED TO THE BACK OF THE MANOR HOUSE, WITH A PILE OF STRAW OVER IN THE FAR CORNER, AND A SHOVEL AND AN AXE LYING UNDERNEATH THE WINDOW. A LARGE LOAF OF BREAD IS ON THE TABLE, AND BESIDE IT IS A NOTE. DOORS LEAVE TO THE NORTH AND TO THE WEST – and you'll find the environment takes on quite solid dimensions in your mind.

Once the map becomes more complex, and the descriptions help clarify the mental images of the rooms, you'll find you have a counterfeit reality with immense power in your hands.

You might like to try and write a simple program, before proceeding further, which allows you to move around the five-room environment we've been looking at.

The adventure program we'll develop in this chapter is a simple one, but it will test your powers of deduction and map-making. Entering and running the program – followed by a careful examination of the listing – should teach you enough about creating adventure games to go ahead and create some brilliant masterpieces of your own.

In this program, *Magician's Maze,* a mean old magician has trapped you in an abandoned house which is littered with magical and valuable items. To secure your freedom, you must transport six different objects into the store room.

Easy, you say, until the magician points out the true nature of his test. The house is built like a maze, and you do not have a map. You have to try and work out the ground plan as you go along. To complicate matters further, you can only carry one thing at a time. Although you can drop any of the objects if you choose, you cannot drop them in a room (except for the store room) which already contains an object.

Sounds complicated? It is, but the satisfaction you'll experience when you solve the maze and complete the task set by the magician will make it more than worthwhile.

The program understands the words you type in (after a limited fashion) and can carry out your wishes. Here's how the program begins:

```
THIS IS MOVE 1

YOU ARE IN THE LIBRARY

THERE ARE EXITS TO THE
      EAST

THE ROOM HOLDS A WILLOW WAND

YOUR ARMS ARE EMPTY
```

```
WHAT DO YOU WANT TO DO? GO EAST
```

As you can see, in common with many computer adventures, you type in your instructions in the form of two words, such as those used here: GO EAST

Moving east, as instructed, you enter the Audience Chamber:

```
THIS IS MOVE 2

YOU ARE IN THE AUDIENCE CHAMBER

THERE ARE EXITS TO THE
      SOUTH EAST WEST

THE ROOM HOLDS A BLACK ACONITE

YOUR ARMS ARE EMPTY

WHAT DO YOU WANT TO DO? GET BLACK
```

Here you see 'black aconite', and decide to pick it up, which you do with the command GET BLACK. In fact, only the first three letters are needed, so GET BLA would have worked as well (and you could have typed in GET BLACK ACONITE if you'd really wanted to do so).

If you are sensible when you play this adventure, you'll be making a map as you work out the relationship of the rooms to each other, so you can move through the maze with growing confidence.

You have picked up the black aconite:

```
THIS IS MOVE 3

YOU ARE IN THE AUDIENCE CHAMBER

THERE ARE EXITS TO THE
      SOUTH EAST WEST

YOU ARE CARRYING A BLACK ACONITE
```

```
WHAT DO YOU WANT TO DO? GO EAST
```

You soon discover your physical limitations as you try to carry two objects at once:

```
THIS IS MOVE 5

YOU ARE IN THE GLOOMY TURRET

THERE ARE EXITS TO THE
      SOUTH WEST

THE ROOM HOLDS A GOLD PIECE

YOU ARE CARRYING A BLACK ACONITE

WHAT DO YOU WANT TO DO? GET GOLD

YOU CAN ONLY CARRY ONE THING AT ONCE
```

So you decide to drop the aconite:

```
THIS IS MOVE 6

YOU ARE IN THE GLOOMY TURRET

THERE ARE EXITS TO THE
      SOUTH WEST

THE ROOM HOLDS A GOLD PIECE

YOU ARE CARRYING A BLACK ACONITE

WHAT DO YOU WANT TO DO? DROP BLACK

THIS ROOM ALREADY
HOLDS A GOLD PIECE
```

Even that is not allowed! Soon, however, you come to the store room:

```
THIS IS MOVE 8

YOU ARE IN THE STORE ROOM
WHICH HOLDS:
      NOTHING

THERE ARE EXITS TO THE
       EAST

YOU ARE CARRYING A BLACK ACONITE

WHAT DO YOU WANT TO DO? DROP BLACK


THIS IS MOVE 9

YOU ARE IN THE STORE ROOM
WHICH HOLDS:
BLACK ACONITE

THERE ARE EXITS TO THE
       EAST

YOUR ARMS ARE EMPTY

WHAT DO YOU WANT TO DO? GO EAST
```

After much wandering about and carrying, you'll find the store room is becoming gratifyingly full:

```
THIS IS MOVE 33

YOU ARE IN THE STORE ROOM
WHICH HOLDS:
BLACK ACONITE
GOLD PIECE
SHINING EMERALD
GLOWING ARTHAME

THERE ARE EXITS TO THE
       EAST
```

```
YOUR ARMS ARE EMPTY

WHAT DO YOU WANT TO DO? GO WEST

ONLY GHOSTS CAN MOVE THROUGH WALLS
```

And so the game continues until you have six objects in the store room. Note that the MSX machine will reject any input from you that it does not understand. To acquire an object, you tell the computer to GET or TAKE it, and to get rid of anything you either DROP it or PUT it. To give up (as if you would!), enter QUIT. GO, followed by NORTH, SOUTH, EAST or WEST will move you in the direction of your choice.

Here's the listing so you can take on the challenge posed by the magician:

```
10 REM MAGICIAN'S MAZE
20 GOSUB 1050:REM INITIALISE
30 FOR Y=1 TO 1000:NEXT Y
40 CLS:PRINT:PRINT
50 MVE=MVE+1
60 GOSUB 1520
70 PRINT TAB(2+RND(1)*6);"THIS IS MOV
E"MVE
80 PRINT:PRINT "YOU ARE IN THE ";N$(R
O)
90 IF RO<>1 THEN 170
100 FLAG=0
110 PRINT "WHICH HOLDS:"
120 FOR Z=1 TO 8
130 IF P$(Z)<>"" THEN PRINT P$(Z):FLA
G=1
140 NEXT Z
150 IF FLAG=0 THEN PRINT TAB(6);"NOTH
ING"
160 PRINT
170 PRINT:PRINT "THERE ARE EXITS TO T
HE"
180 PRINT TAB(7);
190 IF A(RO,1)<>0 THEN PRINT "NORTH "
```

```
;
200 IF A(RO,2)<>0 THEN PRINT "SOUTH "
;
210 IF A(RO,3)<>0 THEN PRINT "EAST ";
220 IF A(RO,4)<>0 THEN PRINT "WEST";
230 PRINT:PRINT
240 IF RO<>1 AND O$(RO)<>"" THEN PRIN
T "THE ROOM HOLDS A ";O$(RO):PRINT
250 IF M$<>"" THEN PRINT "YOU ARE CAR
RYING A ";M$
260 IF M$="" THEN PRINT "YOUR ARMS AR
E EMPTY"
270 GOSUB 310
280 GOTO 30
290 :
300 REM INPUT
310 PRINT:PRINT "WHAT DO YOU WANT TO
DO";
320 INPUT E$
330 GOSUB 1520
340 IF ASC(E$)>ASC("Z") THEN PRINT "E
NGAGE CAPS LOCK":GOTO 320
350 IF LEN(E$)<7 THEN E$=E$+" ":GOTO
350
360 F$=LEFT$(E$,3)
370 IF F$="QUI" THEN 1010
380 IF F$="GO " THEN F$=MID$(E$,4,3):
REM NOTE SPACE WITHIN QUOTE MARKS
      AFTER THE WORD GO
390 IF F$="NOR" OR F$="SOU" OR F$="EA
S" OR F$="WES" THEN GOSUB 490:RETURN
400 Z=1
410 IF MID$(E$,Z,1)=" " THEN G$=MID$(
E$,Z+1,3):GOTO 450
420 IF Z<LEN(E$) THEN Z=Z+1:GOTO 410
430 PRINT:PRINT "I DON'T UNDERSTAND '
";E$;"'"
440 RETURN
450 IF F$="GET" OR F$="TAK" THEN GOSU
B 630:RETURN:REM GET OR TAKE
460 IF F$="PUT" OR F$="DRO" THEN GOSU
```

```
B 700:RETURN:REM PUT OR DROP
470 GOTO 430
480 :
490 REM MOVE
500 F$=LEFT$(F$,1)
510 IF F$="N" THEN 560
520 IF F$="S" THEN 580
530 IF F$="E" THEN 600
540 IF A(RO,4)=0 THEN PRINT "ONLY GHO
STS CAN MOVE THROUGH WALLS":RETURN
550 RO=A(RO,4):RETURN
560 IF A(RO,1)=0 THEN PRINT "THERE IS
 NO EXIT THAT WAY":RETURN
570 RO=A(RO,1):RETURN
580 IF A(RO,2)=0 THEN PRINT "THAT'S A
 SOLID WALL":RETURN
590 RO=A(RO,2):RETURN
600 IF A(RO,3)=0 THEN PRINT THAT IS I
MPOSSIBLE!":RETURN
610 RO=A(RO,3):RETURN
620 :
630 REM GET OBJECTS
640 IF RO=1 THEN RETURN
650 IF LEFT$(G$,3)<>LEFT$(O$(RO),3) T
HEN PRINT "IT IS NOT HERE":RETURN
660 IF M$<>"" THEN PRINT "YOU CAN ONL
Y CARRY ONE THING AT ONCE":RETURN
670 M$=O$(RO)
680 O$(RO)=""
690 RETURN
700 REM PUT THINGS DOWN
710 IF M$="" THEN PRINT "YOU'RE NOT C
ARRYING IT":RETURN
720 IF RO=1 THEN 780
730 IF O$(RO)<>"" THEN PRINT "THIS RO
OM ALREADY":PRINT "HOLDS A ";O$(RO):R
ETURN
740 O$(RO)=M$
750 M$=""
760 RETURN
770 :
```

```
780 PLACE=0:FLAG=0
790 FOR Z=1 TO 6
800 IF PLACE=1 THEN 820
810 IF P$(Z)="" THEN P$(Z)=M$:PLACE=1
:M$=""
820 IF P$(Z)<>"" THEN FLAG=FLAG+1
830 NEXT Z
840 IF FLAG=0 THEN PRINT TAB(8);"NOTH
ING"
850 PRINT 6-FLAG"OBJECTS TO GO"
860 FOR KI=1 TO 700:NEXT KI
870 IF FLAG=6 THEN 900:REM END OF
      GAME, ALL OBJECTS PLACED
880 RETURN
890 :
900 REM END OF GAME
910 PRINT:PRINT "YOU'VE DONE IT"
920 GOSUB 1520:GOSUB 1520
930 PRINT:PRINT "YOU GOT 6 OBJECTS IN
TO THE STORE ROOM"
940 FOR Z=1 TO 8
950 IF P$(Z)<>"" THEN PRINT TAB(2);P$
(Z)
960 NEXT Z
970 PRINT "IT TOOK YOU";
980 PRINT MVE"MOVES"
990 END
1000 :
1010 PRINT:PRINT "I DID NOT THINK YOU
 WERE A QUITTER!"
1020 PRINT:PRINT "YOU SURVIVED FOR";
1030 GOTO 980
1040 :
1050 REM INITIALISATION
1060 SEED=RND(-TIME)
1070 COLOR 15,5:CLS:KEY OFF
1080 DIM A(9,4),N$(9),O$(18),P$(9),M(
8)
1090 RO=INT(RND(1)*8)+2:REM STARTING
                                ROOM
1100 REM NAMES OF ROOMS
```

```
1110 FOR J=1 TO 9
1120 GOSUB 1520
1130 READ N$(J)
1140 NEXT J
1150 REM NAMES OF OBJECTS
1160 FOR J=1 TO 18
1170 READ O$(J)
1180 NEXT J
1190 REM DISTRIBUTE OBJECTS
1200 REM MOSES/OAKFORD ROUTINE
1210 FOR J=18 TO 11 STEP -1
1220 Z=INT(RND(1)*J)+1
1230 M$=O$(Z)
1240 O$(Z)=O$(J)
1250 O$(J)=M$
1260 NEXT J
1270 M$="":REM THIS IS OBJECT
                BEING CARRIED
1280 MVE=0
1290 FOR J=1 TO 9
1300 FOR K=1 TO 4
1310 READ A(J,K)
1320 NEXT K
1330 NEXT J
1340 GOSUB 1520
1350 RETURN
1360 :
1370 DATA "STORE ROOM","GLOOMY TURRET
","MARBLE HALL","LIBRARY","AUDIENCE C
HAMBER","MASTER BEDROM","STUDY","ARTI
ST'S STUDIO","MUSICIAN'S QUARTERS"
1380 DATA "BROKEN CANDLESTICK","GOLD
PIECE","SHINING EMERALD","BRONZE STAT
UE"
1390 DATA "SILVER CHALICE","MIGHTY SW
ORD","DIAMOND RING","CRYSTAL GOBLET",
"SHADOW BOWL"
1400 DATA "MYSTIC SCROLL","WILLOW WAN
D","NEWT'S TAIL","HEMLOCK POTION"
1410 DATA "PAPYRUS PARCHMENT","BLACK
ACONITE","GLOWING ARTHAME","GLASS FRA
```

```
GILUS","POWERFUL PENTACLE"
1420 DATA 0,0,2,0:REM ROOM 1
1430 DATA 0,6,0,1:REM ROOM 2
1440 DATA 0,9,7,0:REM ROOM 3
1450 DATA 0,0,5,0:REM ROOM 4
1460 DATA 0,7,6,4:REM ROOM 5
1470 DATA 2,0,0,5:REM ROOM 6
1480 DATA 5,0,8,3:REM ROOM 7
1490 DATA 0,0,9,7:REM ROOM 8
1500 DATA 3,0,0,8:REM ROOM 9
1510 :
1520 REM MUSIC ROUTINE
1530 SOUND 8,15:SOUND 9,15
1540 NE=INT(RND(1)*150)+10
1550 TWO=INT(RND(1)*90)+NE
1560 FOR NOISE=TWO TO NE STEP-1
1570 SOUND 0,NOISE:SOUND 2,NOISE/2
1580 NEXT NOISE
1590 SOUND 8,0:SOUND 9,0
1600 RETURN
```

Once you've solved the maze as given in the listing, you can tackle two new problems. These new puzzles use the same program, but with different DATA statements for the rooms. Just substitute the following DATA lines for those in the original listing, and you'll have two brand new problems to solve.

Here's the DATA for maze number two:

```
1420 DATA 5,0,6,0:REM ROOM 1
1430 DATA 7,0,0,0:REM ROOM 2
1440 DATA 0,6,4,5:REM ROOM 3
1450 DATA 0,8,0,3:REM ROOM 4
1460 DATA 0,1,3,0:REM ROOM 5
1470 DATA 3,7,0,1:REM ROOM 6
1480 DATA 6,2,0,9:REM ROOM 7
1490 DATA 4,0,0,0:REM ROOM 8
1500 DATA 0,0,7,0:REM ROOM 9
```

And for number three:

```
1420 DATA 5,0,0,8:REM ROOM 1
```

```
1430 DATA 0,4,3,0:REM ROOM 2
1440 DATA 0,7,6,2:REM ROOM 3
1450 DATA 2,8,9,0:REM ROOM 4
1460 DATA 1,0,0,9:REM ROOM 5
1470 DATA 0,0,0,3:REM ROOM 6
1480 DATA 3,0,0,0:REM ROOM 7
1490 DATA 4,0,1,0:REM ROOM 8
1500 DATA 0,0,5,4:REM ROOM 9
```

By all means, change the names of the objects, and of the rooms if you like, and gradually convert this mini-adventure into one of your own.

Once you're fairly confident you know how this adventure was constructed, you can use the program you now have on tape to create other adventures of your own, using the program as a master 'command handler' into which you can pour your own map, room descriptions and monster and treasure names.

Support and Ideas

There is an impressive body of support literature for adventure gaming, as a visit to your local games shop will demonstrate. The selection which follows is very much a product of my own interest in the field, and it should not be seen as even an attempt at selecting the 'best' publications. However, the list is made up from those works which I've found of interest and value. There are probably over a hundred of equal worth, but at least this list should give you a starting point:

THROUGH DUNGEONS DEEP: A Fantasy Gamers' Handbook - Robert Plamondon (Reston Publishing Company, Inc., Reston, Virginia, 1982).

What is Dungeons and Dragons® - John Butterfield, Philip Parker and David Honigmann (Penguin Books Ltd., Harmondsworth, Middlesex, England, 1982). * Dungeons and Dragons is the adventure game which was originated by TSR Hobbies Inc.™ The term is a registered trademark.

Dicing with Dragons, an Introduction to Role-Playing Games – Ian Livingstone (Routledge & Kegan Paul, London, Melbourne and Henley, 1982).

Fantasy Role Playing Games – J. Eric Holmes (Hippocrene Books, Inc., New York, 1981).

As well as books, there are a number of game-playing aids which will help you in building the framework environment within which your adventure can unfold. These include:

DUNGEONS & DRAGONS® boxed game sets are a good way to learn some of the possibilities of adventure gaming. You can start (and stay, if you like) with the BASIC SET WITH INTRODUCTORY MODULE. This set typically contains two books, and a set of six dice, with various numbers of sides. The books are BASIC RULES which sets out the concepts of role-playing, and explains how characters are 'rolled up', how their personalities are derived, how fights are sorted out, and many aspects of the Dungeon or Game Master's role.

As well as dice and BASIC RULES, the set contains a campaign background, THE KEEP ON THE BORDERLANDS. This comes with a great deal of information, including a series of maps, room and player information, and further details on fight resolution. I feel that the BASIC RULES boxed set is probably the best source of ideas you can get your hands on. It is also a very good way to achieve greater understanding of how role-playing games are developed and controlled.
WARNING: These sources of ideas are suggested, of course, only for your own use, to produce games for your own entertainment. You cannot incorporate copyright material into programs for public sale.

The following people make and distribute adventure artifacts and games which may give you further ideas:

Avalon Hill Games, 65Ø High Street, North Finchley, London N12 ØNL.

Citadel Miniatures, 1Ø Victoria Street, Newark, Nottinghamshire.

Games of Liverpool, 5Ø-54 Manchester Street, Liverpool, L1 6ER.

Flying Buffalo, PO Box 1ØØ, Bath Street, Walsall, West Midlands.

Games Workshop Ltd., 1 Dalling Road, London, W6; 27-29 Sunbeam Road, London NW1Ø.

Simpubs Ltd., Oakfield House, 6Ø Oakfield Road, Altrincham, Cheshire WA15 8EW.

TSR Hobbies (UK) Ltd., The Mill, Rathmore Road, Cambridge, CB1 4AD.

CHAPTER SEVENTEEN

Structured Programming Techniques

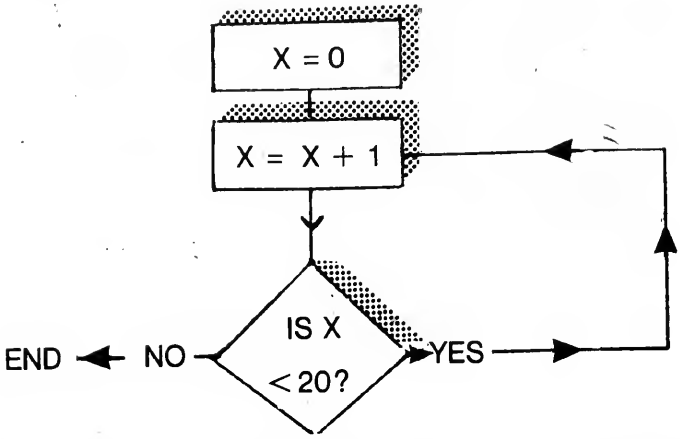
There will come a point in your development as a programmer when you'll have mastered the use of much of the MSX language, and can now concentrate on writing better programs; programs which work after relatively little debugging, which are easy for others to understand and operate, and which are written logically and elegantly.

Your programs will be more likely to run first time if they are planned out carefully before you start entering code (i.e. program) into your computer.

A good way to start is to use a diagram which is often a 'flow chart'. A flow chart is a series of boxes and other shapes, joined by lines, which show the flow of action and decision-making within the computer while the program is running.

The shapes are not too important, and I suggest you stick to just two: a rectangle for most actions the computer must carry out, with a diamond shape each time the computer has to make a decision. The corners of the diamond can be used – as you can see in the diagram – to cater for the alternatives facing the computer.

The diagram shows the flow chart of a program which sets the variable X equal to zero, then adds one to it. The value of X is checked. If X is found to be less than 2Ø, the program goes back to add one to X again. This continues until the value of X equals 2Ø.



One advantage of using a flow chart is that you do not get locked, at an early stage of your work, into the peculiarities of the language, its weaknesses and strengths, which you are using. Instead, you concentrate on what you want to do.

A flow chart is 'universal'. The same flow chart can be used as the basis of a program written on a computer furnished with a completely different language from the one in which you originally intended to write the program.

A flow chart models the flow of action and logic within a program, and is therefore very useful for picking up potential bugs at the earliest stages. You may, for example, find that one condition the program will test will never be fulfilled, possibly leading to the program being trapped in an infinite loop. Other parts of the code may be bypassed completely, because the condition which triggers entry into that part of the code will never be met.

Once you've devised a flow chart for your program, and you've run through it mentally a few times, so that the most obvious bugs are removed, you should reduce the flow chart to a series of subroutine calls. Although it seems pretty silly to do this for a simple program like our "SET X EQUAL TO ZERO, ADD ONE, CHECK IF LESS THAN 20" program, this method comes into its own with more complex programs.

Programming in Modules

As you've seen in many programs in this book, I'm in favour of you starting your program with a series of subroutine calls, with each action of the program being looked after by a separate subroutine. Then, if the steps within a program have to be performed several times in a particular sequence, the series of subroutine calls can be cycled through – over and over again – until a particular condition is met which signals the end of the run.

You'll recognize how useful this approach to programming can be when you get to the debugging stage of your program. If there is a bug, it is likely to be within a single subroutine, so it will be relatively easy to pin down the subroutine which contains the bug, rather than having to work right through the program trying to track it down.

Working with subroutine 'modules' in this way allows you to *test* sections of the program in isolation, even before the entire program is finished. I'll try to make this statement clear by showing you the first part of a typical program to play Checkers:

```
10 REM CHECKERS
20 GOSUB 9000:REM INITIALISE
30 GOSUB 8000:REM PRINT BOARD
40 GOSUB 7000:REM ACCEPT PLAYER MOVE
50 GOSUB 8000:REM PRINT BOARD
60 GOSUB 5000:REM MSX MACHINE MOVES
70 IF (HUMAN NOT WON) AND (COMPUTER
      NOT WON) THEN 30
80 IF (COMPUTER HAS WON) THEN
      PRINT "I WIN"
90 IF (HUMAN HAS WON) THEN
      PRINT "YOU WIN"
100 END
```

You could have quite a bit of this program running, and tested (such as the initialisation routine, printing the board and accepting the player's move), before you even turned your attention to how on earth you were going to get the computer to make its move.

You would then know – for example – that you would not need to

waste any extra thought on whether or not an error in the board-printing routine was the cause of odd output. Having tested the board subroutine and the player move routine, you'd know that the error must be in the subroutine between lines 5000 and 6999, the subroutine in which the computer makes its move.

All you need to do is put a single PRINT statement, such as "THIS IS COMPUTER MAKING A MOVE" followed by a RETURN for incomplete subroutines, knowing that the program will accept that, and demonstrate the direction the program flow is following, even if whole sections of code have not yet been written.

In general, I'd advise you to use this system of using a 'master loop' of subroutine calls within which you will 'hold' the entire program.

I suggest you try and do as much writing of the program as possible *before* you turn the computer on, even though there is a great temptation to dive straight into the computer and start punching in code. You'll find that the discipline of writing it out by hand in advance will serve you in good stead and should, in the long run, produce a better program than might otherwise have been the case.

Overall, you'll probably end up spending less time on the program working in this way than you would if you began the process sitting at the computer keyboard.

It took me a while to learn this lesson. Although I had read suggestions along the lines of 'work out exactly what you're going to enter *before* you start at the computer' in several books, I tended to just jump right in without much prior thought.

Although I worked out rough flow charts, and had an idea what sort of display organisation I wanted, I certainly did not write much program out on paper before starting at the computer. Then, I once found myself stuck for a two-week period without a computer and ideas for several programs just itching to be written. I had to write them out in an exercise book.

The relative ease with which the programs were debugged when they were eventually entered into the computer, and the complexity of the programs I wrote in this way (including my first chess program)

convinced me that this was the way I would work from then on. It is amazing how much cleaner a program can be if all the rough working out is done on paper, rather than on the computer screen.

When working on programs, I tend to write the major 'call subroutines loop' first of all, but without line numbers, so the program contains lines like GOSUB PRINT BOARD and GOSUB INITIALISE. Next, I write each module (or subroutine) on a separate sheet of paper. Then, when the major subroutines have been written I shuffle them into an order which seems most logical.

All this, of course, occurs before any line numbers are written in. The subroutine modules are put in an order which ensures that the program structure is as logical as possible. I use arrows to indicate the destination of GOTO's within a module, and *names* for subroutines, as suggested in the major loop. Later on, when the program has started to assume a firm shape, the lines are numbered (I always work in tens, starting at line 1Ø) and the relevant GOTO and GOSUB destinations are added.

All programs have an 'end condition' at which point computation stops. It is worth putting a test for this end condition as part of the GOTO which sends the program back to start cycling through the major subroutine loop. This ensures that the cycle will continue until a particular condition is met, at which point the program 'falls through' that GOTO and continues on to the lines which signal the end of the program.

Explicit input prompts and print statements

It is very useful, when writing a program, to keep in mind how the program will appear to a stranger when it is run by him or her for the first time. If there is an *input prompt* required, it is far more useful if the program prints up something like "HOW MANY HOURS HAS THE EMPLOYEE WORKED THIS WEEK?" instead of just "INSERT HOURS?" or the almost useless lone question mark.

The same suggestion applies to print output. It is far better that your program is written so that it prints out THE NUMBER OF HOURS WORKED ON FULL PAY THIS WEEK IS 27, rather than

HOURS, FULL: 27 or an unsupported 27. Of course, providing explicit input prompts and output PRINT statements consumes memory as well as typing time when entering the program, but the contribution they make to the final program means the trouble involved is well worth it.

REM Statements

While exact PRINT statements and input prompts will help a person running a program make sense of it, REM statements can help make the program clear to those who are examining the listing for the first time. REM statements (which are, as you know, ignored by the computer when a program is run) should be used to help illuminate the flow of logic within the program, and what is happening in certain places within it. This is especially important in parts of the program where decisions are made, or calculations are carried out.

Not only can REM statements be used to explain different parts of the code, but they can also be used to provide visual 'breaks', so that the various blocks of code which carry out certain tasks are visually separated from the rest of the program. A blank REM statement (the word REM standing alone in a program line) can be used for this. A row of asterisks is an effective alternative.

Variables

It is worth considering the use of explicit names for variables, using either the word in full (such as HOURS as a variable name in a payroll program for hours worked) or an abbreviated version of this (such as HR) which has a fairly obvious meaning. You'll discover that this makes it easy, when working on a program, to keep track of your variables. This will help you with the initial debugging and later on as well if you need to improve or extend the original program.

Explicit variable names also help make your code more 'transparent' so other programmers can work out what the various parts of the program are intended to do. You'll also find it a great help to yourself when you return to the program at a later date. It is surprising how code which seemed incredibly clear in terms of its

purpose when you entered it, becomes exceedingly dense when you return to it after a long break.

Checking Input

Any input entered by the user should be checked by the program before it is accepted to ensure that incorrect data does not cause the program to crash at some future point. Whether you want string, or numeric input, it is often wise to allow for string input, which is first checked to see whether the entered material is acceptable and then, if necessary, the string can be changed into a number.

For example, if the user needs to enter a number between one and nine, a string can be accepted and then checked to ensure that it is not less than "1" and more than "9" before being changed into a number with a command like VAL.

As well as ensuring that the program will reject invalid input, you should check the program to see all the inputs which it does accept produce sensible answers when processed later on in the program. For example, make sure that your program does not accept zero as a possible number if the computer must later divide by this number.

Similarly, if numbers are to be processed by a function, and then the result of this processing used for division, you must check that an apparently valid input does not turn into zero as a result of evaluation by the function.

If the information entered by the user is rejected, and a new input is requested, the program should ideally point out why the original input was not suitable, or spell out again exactly what is required (such as "ENTER A NUMBER BETWEEN 1 AND 4"). You risk making users angry if input which appears valid is continually rejected without apparent reason.

Documentation

The written material which accompanies a program is often called documentation. It is useful for a program to be supported by some documentation, however sketchy.

The written information should explain, of course, what the program does, then go on to outline the flow of action within the program. The documentation should alert the user as to the kind of actions which will be required from him or her when running the program, and give an indication of the kinds of user input and reaction which will be accepted.

The format of the final output should also be discussed. A list of variable names can be included.

If there are ways in which the program can be developed, extended or improved, suggestions along these lines can be added to the documentation. Written references to any material which will help in understanding the algorithms used, or for giving suggested areas for program development, should also be included.

In many ways, it is reasonable to assume that the job of programming is not finished once the program is done. Without documentation, the job is only three-quarters complete. Documentation finishes the task, adding a professional stamp to your work which allows the program you've written to be used most effectively.

Possibly the only time extensive documentation is not really required is when the program is 'menu-driven'. A program which uses REM statements extensively may not need very much in the way of documentation, especially if you can include a variable list, as a series of REM statements, at the end of the program.

Generally, however, you'll find it better to document a program externally, rather than rely on REM statements, or the various menu choices, to do the job for you. It is worth trying to write your program documentation so that it would make sense to someone who has not seen the program running.

This person should be able to get a very good idea, just from reading the documentation, of what the program does and how it does it; how it interacts with the user both in terms of accepting information and in presenting the results of its computations to the users; and how the program is organised as a whole.

Documentation for a major program should start with an

introduction which quickly explains what is going on, and tells how to use the program. The later parts of the documentation can then discuss the program in greater detail. It is not good practice to force the user to wade through a vast amount of information in order to dig out the vital facts he or she needs to get the program running.

APPENDIX

Computer Terms

Accumulator – part of the computer's logic unit which stores the intermediate results of computations.

Address – a number which refers to a location, generally in the computer's memory, where information is stored.

Algorithm – the sequence of steps used to solve a problem.

Alphanumeric – generally used to describe a keyboard, and signifying that the keyboard has alphabetical and numerical keys. A numeric keypad, by contrast, only has keys for the digits one to nine, with some additional keys for arithmetic operations, much like a calculator.

APL – this stands for Automatic Programming Language, a language developed by Iverson in the early 196Øs, which supports a large set of operators and data structures. It uses a non-standard set of characters.

Application software – these are programs which are tailored for a specific task, such as word processing, or to handle mailing lists.

Artificial Intelligence – the section of computer science which concentrates on eliciting machine behaviour which, if it came from a human being, would be called intelligent. One of the aims of AI (as Artificial Intelligence is often written) is to make computers more useful. AI research may also turn out to help us understand our own thinking processes.

ASCII – this stands for American Standard Code for Information Exchange. This is an almost universal code for letters, numbers and

symbols, which has a number between Ø and 255 assigned to each of these, such as 65 for the letter A.

Assembler – this is a program which converts another program written in an assembly language (which is a computer program in which a single instruction, such as ADD, converts into a single instruction for the computer) into the language the computer uses directly.

BASIC – stands for Beginner's All-purpose Symbolic Instruction Code, the most common language used on microcomputers. It is easy to learn, with many of its statements being very close to English. MSX BASIC is a version of BASIC.

Batch – a group of transactions which are to be processed by a computer in one lot, without interruption by an operator.

Baud – a measure of the speed of transfer of data. It generally stands for the number of bits (discrete units of information) per second.

Benchmark – a test which is used to measure some aspect of the performance of a computer, which can be compared to the result of running a similar test on a different computer.

Binary – a system of counting in which there are only two symbols, 'Ø' and '1' (as opposed to the ordinary decimal system, in which there are ten symbols, 'Ø', '1', '2', '3', '4', '5', '6', '7', '8' and '9'). Your computer 'thinks' in binary.

Boolean Algebra – the algebra of decision-making and logic, developed by English mathematician George Boole, and at the heart of your computer's ability to make decisions.

Bootstrap – a program, run into the computer when it is first turned on, which puts the computer into the state where it can accept and understand other programs.

Buffer – a storage mechanism which holds input from a device such as keyboard, then releases it at a rate which the computer dictates.

Bug – an error in a program.

Bus – a group of electrical connections used to link a computer with an ancillary device, or another computer.

Byte – the smallest group of bits which makes up a computer word. Generally a computer is described as being 'eight bit' or '16 bit', meaning the word consists of a combination of eight or sixteen zeros or ones.

Central Processing Unit (CPU) – the heart of the computer, where arithmetic, logic and control functions are carried out.

Character code – the number in ASCII (see ASCII) which refers to a particular symbol, such as 32 for a space and 65 for the letter 'A'.

COBOL – stands for Common Business Orientated Language, a standard programming language, close to English, which is used primarily for business.

Compiler – a program which translates a program written in a high level (human-like) language into a machine language which the computer is able to understand directly.

Concatenate – to add (adding two strings together is known as 'concatenation').

CP/M – these initials stand for Control Program/Microcomputer, an almost universal disk operating system developed and marketed by Digital Research, Pacific Grove, California.

Data – a general term for information processed by a computer.

Database – a collection of data, organised to permit rapid access by computer. A relational data base is one in which the interconnections between various elements within the database are stored explicitly to aid manipulation of, and access to, the elements within the data base.

Debug – to remove bugs (errors) from a program.

Disk – a magnetic storage medium (further described as a 'hard disk', 'floppy disk' or even 'floppy') used to store computer

information and programs. The disks resemble, to a limited extent, 45 rpm sound records, and are generally eight, five and a quarter, or three and a half inches in diameter. Smaller 'microdisks' are also available for some systems.

Documentation – the written instructions and explanations which accompany a program.

DOS – stands for Disk Operating System (and generally pronounced 'doss'), the versatile program which allows a computer to control a disk system.

Dot-matrix printer – a printer which forms the letters and symbols by a collection of dots, usually on an eight by eight, or seven by five, grid.

Double-density – adjective used to describe disks when recorded using a special technique which, as the name suggests, doubles the amount of storage the disk can provide.

Dynamic memory – computer memory which requires constant recharging to retain its contents.

EPROM – stands for Erasable Programmable Read Only Memory, a device which contains computer information in a semi-permanent form, demanding sustained exposure to ultra-violet light to erase its contents.

Error messages – information from the computer to the user, sometimes consisting only of numbers or a few letters, but generally of a phrase (such as 'Out of memory') which points out a programming or operational error which has caused the computer to halt program execution.

Expert system – a computer program which, drawing on the encoded expertise of human experts, performs a specialised task as well as (or, in some cases, better than) the human expert in that field would. They often call on extensive databases of knowledge, and are sometimes called knowledge-based systems.

Field – a collection of characters which form a distinct group, such as

an identifying code, a name or a date; a field is generally part of a record.

File – a group of related records which are processed together, such as an inventory file or a student file.

Firmware – the solid components of a computer system are often called the 'hardware', the programs, in machine-readable form on disk or cassette, are called the 'software', and programs which are hard-wired into a circuit, are called 'firmware'. Firmware can be altered, to a limited extent, by software in some circumstances.

Flag – this is an indicator within a program, with the 'state of the flag' (i.e. the value it holds) giving information regarding a particular condition.

Floppy disk – see disk.

Flowchart – this is a written layout of program structure and flow, using various shapes, such as a rectangle with sloping sides for a computer action, and a diamond for a computer decision. A flowchart is generally written before any lines of program are entered into the computer.

FORTRAN – a high level computer language, generally used for scientific work (from FORmula TRANslation).

Gate – a computer 'component' which makes decisions, allowing the circuit to flow in one direction or another, depending on the conditions to be satisfied.

GIGO – acronym for 'Garbage In Garbage Out', suggesting that if rubbish or wrong data is fed into a computer, the result of its processing of such data (the output) must also be rubbish.

Global – a set of conditions which affects the entire program is called 'global', as opposed to 'local'.

Graphics – a term for any output of computer which is not alphanumeric, or symbolic.

Hardware – the solid parts of the computer (see 'software' and 'firmware').

Heuristic – a path towards a goal which has been worked out by experience, rather than by calculation; a path of this type is not guaranteed to produce a certain result (in contrast to an algorithm which is a technique or procedure which, when applied, produces a desired result); chess programs play, in large measure, heuristically.

Hexadecimal – a counting system often used by machine code programmers because it is closely related to the number storage methods used by computers, based on the number 16 (as opposed to our 'ordinary' number system which is based on 1Ø).

Hex pad – a keyboard, somewhat like a calculator, which is used for direct entry of hexadecimal numbers.

High-level languages – programming languages which are close to English. Low-level languages are closer to those which the computer understands. Because high-level languages have to be compiled into a form which the computer can understand before they are processed, high-level languages run more slowly than do their low-level counterparts.

Human interface – the 'outward and visible sign' of an expert system, with which the human user of the system interacts; these often work with natural language.

Inference system – the mechanism by which a computer program reaches conclusions; some systems make hard and fast YES/NO decisions, others operate in the world of 'fuzzy logic', where shades of grey (degrees of uncertainty) are permitted.

Input – any information which is fed into a program during execution.

I/O – stands for Input/Output port, a device the computer uses to communicate with the outside world.

Instruction – an element of programming code, which tells the computer to carry out a specific task. An instruction in assembler

language, for example, could be ADD which (as you've guessed) tells the computer to carry out an addition.

Interpreter – converts the high-level ('human-understandable') program into a form which the computer can understand.

Joystick – an analogue device which feeds signal into a computer which is related to the position which the joystick is occupying; generally used in games programs.

Kilobyte – the unit of memory measurement; one kilobyte (generally abbreviated as K) equals 1,Ø24 bits (a bit is the smallest discrete unit).

Knowledge base – the accumulated knowledge upon which an expert system makes its judgements.

Knowledge engineering – the process by which human expertise is transferred to an expert system.

Knowledge Information Processing Systems – these are the Fifth Generation computer systems which are under development in Japan.

Low-level language – a language which is close to that used within the computer (see high-level language).

Machine language – the step below a low-level language; the language which the computer understands directly.

Memory – the device or devices used by a computer to hold information and programs being currently processed, and for the instruction set fixed within a computer which tells it how to carry out the demands of the program. There are basically two types of memory (RAM and ROM).

Microprocessor – the 'chip' which lies at the heart of your computer. This does the 'thinking'.

Modem – this stands for MOdulator/DEModulator, and is a device

which allows one computer to communicate with another via the telephone.

Monitor – (a) a dedicated television-screen for use as a computer display unit, contains no tuning apparatus; (b) the information within a computer which enables it to understand and execute program instructions.

Motherboard – a unit, generally external, which has slots to allow additional 'boards' (circuits) to be plugged into the computer to provide facilities (such as high-resolution graphics, or 'robot control') which are not provided with the standard machine.

Mouse – a control unit, slightly smaller than a box of cigarettes, which is rolled over the desk, moving an on-screen cursor in parallel to select options and make decisions within a program. 'Mouses' work either by sensing the action of their wheels, or by reading a grid pattern on the surface upon which they are moved.

Network – a group of computers working in tandem.

Numeric pad – a device primarily for entering numeric information into a computer, similar to a calculator.

Octal – a numbering system based on eight (using the digits Ø, 1, 2, 3, 4, 5, 6 and 7).

On-line – device which is under the direct control of the computer.

Operating system – this is the 'big boss' program or series of programs within the computer which controls the computer's operation, doing such things as calling up routines when they are needed and assigning priorities.

Output – any data produced by the computer while it is processing, whether this data is displayed on the screen or dumped to the printer, or is used internally.

Pascal – a high level language, developed in the late 1960s by Niklaus Wirth, which encourages disciplined, structured programming.

Port – an output or input 'hole' in the computer, through which data is transferred.

Program – the series of instructions which the computer follows to carry out a predetermined task.

PILOT – a high level language, generally used to develop computer programs for education.

RAM – stands for Random Access Memory, and is the memory on board the computer which holds the current program. The contents of RAM can be changed, while the contents of ROM (Read Only Memory) cannot be changed under software control.

Real-time – when a computer event is progressing in line with time in the 'real world', the event is said to be occurring in real time. An example would be a program which showed the development of a colony of bacteria which developed at the same rate that such a real colony would develop. Many games, which require reactions in real time, have been developed. Most 'arcade action' programs occur in real time.

Refresh – the contents of dynamic memories (see memory) must receive periodic bursts of power in order for them to maintain their contents. The signal which 'reminds' the memory of its contents is called the refresh signal.

Register – a location in computer memory which holds data.

Representation – the organisation of information within a computer so that it can be managed by the knowledge base control system.

Reset – a signal which returns the computer to the point it was in when first turned on.

ROM – see RAM.

RS-232 – a standard serial interface (defined by the Electronic Industries Association) which connects a modem and associated terminal equipment to a computer.

S-100 bus – this is also a standard interface (see RS-232) made up of 100 parallel common communication lines which are used to connect circuit boards within micro-computers.

SNOBOL – a high level language, developed by Bell Laboratories, which uses pattern recognition and string manipulation.

Software – the program which the computer follows (see firmware).

Stack – the end point of a series of events which are accessed on a last in, first out basis.

Subroutine – a block of code, or program, which is called up a number of times within another program.

Symbolic inference – see inference system.

Syntax – as in human languages, the syntax is the structure rules which govern the use of a computer language.

Systems software – sections of code which carry out administrative tasks, or assist with the writing of other programs, but which are not actually used to carry out the computer's final task.

Thermal printer – a device which prints the output from the computer on heat-sensitive paper.

Time-sharing – this term is used to refer to a large number of users, on independent terminals, making use of a single computer, which divides its time between the users in such a way that each of them appears to have the 'full attention' of the computer.

Turnkey system – a computer system (generally for business use) which is ready to run when delivered, needing only the 'turn of a key' to get it working.

VLSI – Very Large Scale Integration of components on a 'chip', with present development in Japan and the US aiming for the equivalent of 10 million transistors per chip (current chips contain the equivalent of around half a million transistors at most).

Volatile memory – a memory device which loses its contents when the power supply is cut off.

Word processor – a dedicated computer (or a computer operating a word-processing program) which gives access to an 'intelligent typewriter' with a large range of correction and adjustment features.

NOTES

NOTES

NOTES

NOTES

Here's your chance to make the most of the exciting sound and graphics on your MSX computer. Tim Hartnell, the most widely-published author of popular microcomputer books in the world, takes you through all the important elements of MSX BASIC, step by simple step.

You'll discover you're learning to program your new computer easily and quickly... and before long, you'll be programming it like a professional.

Save the games and other programs in the book as you go along, and soon you'll have the start of a great library of software.

Programs in this book include:

- **Magician's Maze** (adventure)
- **Monster Chase** (moving graphics)
- **Zap Out** (arcade-like sprite program)
- **MSX-Thello** (Othello)
- **MSX-Checkers**
- **Duck Shoot**

Whether you've never programmed a computer before, or you already know how to program in BASIC and now want to take advantage of the special MSX features, you'll find a great deal of value in this exciting new book.

£6.95

ISBN 0-947695-28-1